第一日　绪　　论

1.1　集合与关系

俗话说："物以类聚"。世界上万事万物，大至宇宙、星系，小至原子、电子，在物质及其运动的各个层次上，各种事物无一不可分门别类。人们把同类的事物放在一起考虑，就组成了所谓的集合。在我们日常生活中，集合的例子比比皆是。例如，"太阳系中行星的全体"是一个集合；"某所学校所有在校学生"是一个集合。总之，当某些具有共同性质的事物汇合在一起，就形成了一个集合。

一、集合

1．集合的概念及其表示法

我们把具有某种共同性质的事物的全体称为集合，简称集。组成集合的每个事物称为这个集合的元素。习惯上用大写字母 A、B、…等表示集合，用小写字母 a、b、…等表示集合的元素。当然，在一些特殊集合中，元素往往有既定符号。在本日中，自然数集用 N 表示；整数集用 Z 表示；有理数集用 Q 表示；实数集用 R 表示；复数集用 C 表示。

若有一个集合 A，当事物 a 是 A 中的一个元素时，就称 a 属于 A，记为 $a\in A$；当事物 a 不是 A 中的一个元素时，就称 a 不属于 A，记为 $a\notin A$。例如：$-2\in Z$，$-2\notin N$。

假定一个集合 A 中包含 n 个元素 $a_1, a_2, \cdots, a_n$ $(n>=0)$，则称集合 A 是一个有限集，记为 $A=\{a_1, a_2, \cdots, a_n\}$。

例如：设 P 是小于 12 的质数集。因为 P 的元素为 2、3、5、7、11，

所以可记为

$$p=\{2,3,5,7,11\}$$

集合既然是由具有某种共同性质的事物组成的，那么用这种性质同样也能限定集合由哪些元素组成。这样就得到了集合的另一种表示方法，其一般形式如下：

$$A=\{x|P\}$$

其中 P 是指某种性质，$x|P$ 就是指元素 x 具有性质 P，而 $A=\{x|P\}$ 则表示集合 A 是由所有具有性质 P 的那些 x 组成的。

例如：方程 $x^2-4x+3=0$ 的解集 S，可表示成

$S=\{1,3\}$　或　$S=\{x|x^2-4x+3=0,x\in R\}$

当集合 A 中不包含任何元素时，则称 A 是一个空集，空集用符号 Φ 表示。如果集合 A 中包含的元素是无限的，则集合 A 称为无限集。在今后的讨论中我们将主要是在有限集上进行，因此如果未作说明，所说的集合都是指有限集。

2. 集合之间的关系和运算

(1) 集合的之间的关系

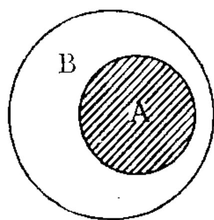


图 1-1　集合包含关系图示

设 A 和 B 是两个集合，如果 A 的每一个元素都属于集合 B，则称集合 B 包含集合 A，或称集合 A 包含于集合 B，记为 $A\subset B$。这时也把集合 A 称为集合 B 的子集，把集合 B 称为集合 A 的扩张集。用 Venn（文氏）图来表示 $A\subset B$ 是非常直观的，如图 1-1 所示。

如果集合 A 是集合 B 的子集，并且 B 中至少有一个元素 a 不属于 A，则称 A 是 B 的真子集。

例如：设集合 A 为 6 的正因数集，B 为 12 的正因数集，则

$A=\{1,2,3,6\}$，

$B=\{1,2,3,4,6,12\}$

因为集合 A 的元素都是集合 B 的元素，所以 $A\subset B$，即 A 是 B 的子集，且是真子集。

如果给定两个集合 A 和 B，它们满足 $A \subset B$ 和 $B \subset A$，则称集合 A 和 B 相等，记为 $A=B$。也就是说两个集合中含有相同的元素(次序不一定一致)。

例如：$C=\{2,3,1\}$　　　　$D=\{1,2,3\}$

显然，由于集合 C 中的元素和集合 D 中的元素相同，即 $C=D$。

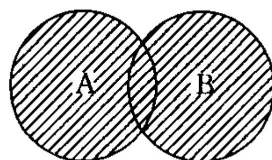
(2) 集合之间的运算

对于集合 A、B，定义它们的并集 $A \cup B$、交集 $A \cap B$ 和差集 $A-B$ 如下：

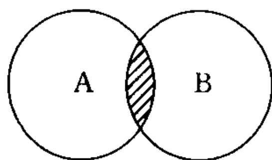
(1) 给定两个集合 A、B，由集合 A 和 B 中的所有元素构成的集合称为 A 和 B 的并集，记为 $A \cup B$。图 1-2(a)为其文氏图表示。

(2) 给定两个集合 A、B，由同时属于集合 A、B 的所有元素构成的集合称为 A 和 B 的交集，记为 $A \cap B$。图 1-2(b)为其文氏图表示。

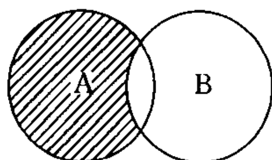
(3) 给定两个集合 A、B，由所有属于集合 A 而不属于集合 B 的那些元素构成的集合称为 A 和 B 的差集，记为 $A-B$。图 1-2(c)为其文氏图表示。



(a) 集合的并($A \cup B$)



(b) 集合的交($A \cap B$)



(c) 集合的差($A-B$)

图 1-2　集合的运算

例如：$A=\{1,2,3,4\}$　　$B=\{1,3,5,7\}$

则　$A \cup B=\{1,2,3,4,5,7\}$

$A \cap B=\{1,3\}$

$A-B=\{2,4\}$

如果集合 A 和 B 满足 $A \cap B=\Phi$，就称集合 A 和 B 是不相交的。按照交集的定义，这也就是说，集合 A 和 B 没有公共的元素。

二、关系

在此我们首先介绍有序对(序偶)的概念。在初等数学中，最常见最典型的序偶，就是平面直角坐标系中点的坐标 $\langle x,y \rangle$。例如：点 $\langle 1,-2 \rangle$、点 $\langle -3,5 \rangle$ 等。如果 a_1, a_2 是两个元素，按先后顺序将它们排列在一起，并且作为一个整体来看待，则称它为一个序偶，记为 $\langle a_1, a_2 \rangle$。

注意$\langle a_1,a_2\rangle$和$\langle a_2,a_1\rangle$是不同的，因为它们的排列顺序不同。例如，在平面直角坐标系中$\langle 1,-2\rangle$和$\langle -2,1\rangle$是不同的，它们代表两个不同的点。

如果给定两个集合A和B，则定义它们的笛卡尔积如下，记为A×B。

$$A\times B=\{\langle a,b\rangle \mid a\in A, b\in B\}$$

例如：设$A=\{a,b\}$，$B=\{1,2,3\}$，那么A和B的笛卡尔积为：

$$A\times B=\{\langle a,1\rangle,\langle a,2\rangle,\langle a,3\rangle,\langle b,1\rangle,\langle b,2\rangle,\langle b,3\rangle\}$$

$$B\times A=\{\langle 1,a\rangle,\langle 1,b\rangle,\langle 2,a\rangle,\langle 2,b\rangle,\langle 3,a\rangle,\langle 3,b\rangle\}$$

一般地，对于任何两个有限集A和B来讲，如果A有n个元素，B有m个元素，那么，A×B和B×A都有n×m个元素。但是，正如上例中所表明的，A×B和B×A的元素个数虽然一样，其元素却不同，它们通常是两个不同的集合，即$A\times B\neq B\times A$。在A×B中，如果B和A相等，即B=A，则笛卡尔积自然应为A×A。

集合A×B的每个子集R都称为从A到B的一个关系，或者称R是A×B的一个关系。当A=B时，就称R是定义在A上的一个关系。如果R是定义在集合A上的一个关系，则$\langle a,b\rangle\in R$时，就称元素a和b有关系R，记作aRb。此时元素a称为元素b的前缀（或前驱），元素b称为元素a的后继。

设R是非空集合A上的一个关系，如果对于A中的任何元素a，都有$\langle a,a\rangle\in R$，则称关系R是自反的。如果对于A中任何元素a，都没有$\langle a,a\rangle\in R$，则称关系R是反自反的。如果对于A中的任何元素a和b，当$\langle a,b\rangle\in R$时，必有$\langle b,a\rangle\in R$，则称关系R是对称的。如果$\langle a,b\rangle\in R$且$\langle b,a\rangle\in R$时，必有a=b，则称R是反对称的。如果对于A中的任何元素a、b、c，当$\langle a,b\rangle\in R$，并且$\langle b,c\rangle\in R$，必有$\langle a,c\rangle\in R$时，则称关系R是传递的。

例如：整数集Z上的关系“=”是自反的，因为对于任意$x\in Z$，总有x=x；而整数集Z上的关系“<”是反自反的。因为对于任何$x\in Z$，x<x都不成立。整数集Z上的关系“=”是对称的，而关系“>”、“≥”、“<”、“≤”是反对称的。整数集Z上的关系“<”是传递的，因

为对于任何 a、b、c∈Z，如果 a<b 且 b<c，则必有 a<c。

当集合 A 上定义的关系 R 是自反的、对称的和传递的，这时就称 R 是一个等价关系。此时，如果⟨a,b⟩∈R，就称元素 a 和 b 是等价的。例如，整数集 Z 上的关系“＝”是一个等价关系。

当集合 A 上定义了一个等价关系 R 时，可以根据这个关系将 A 中的元素分成若干部分，让它们分别属于 A 的若干个子集，这些子集互不相交，并且使同一个子集的任何两个元素都具有关系 R，而任何两个子集中的任何两个元素，都不满足关系 R，即这些子集构成了集合 A 的一些划分，而这些子集都称为关系 R 的等价类。

当集合 A 上定义的关系 R 是自反的、反对称的和传递的，则称 R 是集合 A 上的一个部分序(或偏序)关系。如果在集合 A 上定义了一个偏序关系 R，则称集合 A 为偏序集，记为(A,R)。显然，对于一个有限偏序集来说，至少有一个元素没有前缀，至少有一个元素没有后继。例如：自然数集 N、整数集 Z、有理数集 Q 和实数集 R 上的关系“≤”就是一个偏序关系。又如：在自然数集 N 上定义关系“/”如下：⟨x,y⟩∈/当且仅当 x 整除 y。不难证明，/是自然数集 N 上的一个偏序关系。

如果 R 是定义在集合 A 上的偏序关系，即(A,R)是一个偏序集，并且关系 R 满足下面条件：对于 A 中的任何元素 a、b，⟨a,b⟩∈R 和⟨b,a⟩∈R 两者至少有一个成立，这时就称 R 是集合 A 上的一个序(或完全序)关系，A 在这个序关系下称为有序(或完全序)集，仍记为(A,R)。例如：自然数集 N、整数集 Z、有理数集 Q 和实数集 R 在关系“≤”下都是完全序集。而自然数集 N 在如上定义的关系“/”下就不是完全序集。

1.2 数据结构概念

“数据结构”是一门随着计算机科学的发展而逐渐形成的学科，自 1946 年美国宾夕法尼亚大学的工程师和科学家发明了第一台电子计算机以来，计算机科学经历了将近半个世纪的蓬勃发展，其发展

速度远远超出了人们的预料，可谓日新月异。计算机硬件技术，软件技术不断提高，使计算机价格越来越便宜，功能越来越强大，也就使得其应用领域越来越广泛。计算机的应用早已不再局限于科学计算，而更多地用于过程控制、数据处理、信息管理，以及计算机辅助设计(CAD)和计算机辅助制造(CAM)等非数值数据的处理工作。与此相应，计算机加工处理的对象——数据也从单纯的数值数据发展到字符、表格、图像以及声音等各种非数值数据。为了更有效地使用计算机，设计出高效、可靠的程序，需要对数据的组织、数据元素之间的关系、数据在计算机中的表示(包括数据元素的表示和数据元素之间关系的表示)以及对数据的操作进行深入研究。这样，就促进"数据结构"这门学科的形成和发展。

一、什么是数据结构

在介绍数据结构这一概念之前，首先要了解一下数据的概念。所谓数据就是指对客观事物的符号表示。在计算机科学中我们把所有能输入到计算机中，并能被计算机处理的符号总称为数据。换言之，它就是计算机程序加工的原料。例如，一个代数方程求解程序，它处理的对象(数据)就是实数；又如：一个文字处理程序(如 WPS)，其处理的对象就是字符文字。对于计算机科学而言，数据的含义极其广泛，图像、声音等都可以通过编码而归之为数据的范畴。数据由数据元素组成，数据元素是数据的基本单位，通常在计算机程序中作为一个整体进行考虑和处理。而数据元素之间往往又不是孤立无关的，数据结构就是相互之间存在一种或多种特定关系的数据元素的集合，数据元素之间存在的相互关系就叫结构。根据数据元素之间关系的不同特性，通常有下列四类基本结构：

1. 集合：结构中的数据元素之间除了"同属于一个集合"的关系外，别无其它关系；

2. 线性结构：结构中的数据元素之间存在一个对一个的关系；

3. 树形结构：结构中的数据元素之间存在一个对多个的关系；

4. 图状结构(网状结构)：结构中的数据元素之间存在多个对多

个的关系。

图 1-3 为上述四类基本结构的关系图。其中,圆圈表示数据元素,圆圈之间的连线表示数据元素之间的关系。在本书中将主要讨论线性结构、树形结构和网状结构。“集合”,由于元素之间的关系极为松散,所以不作讨论。

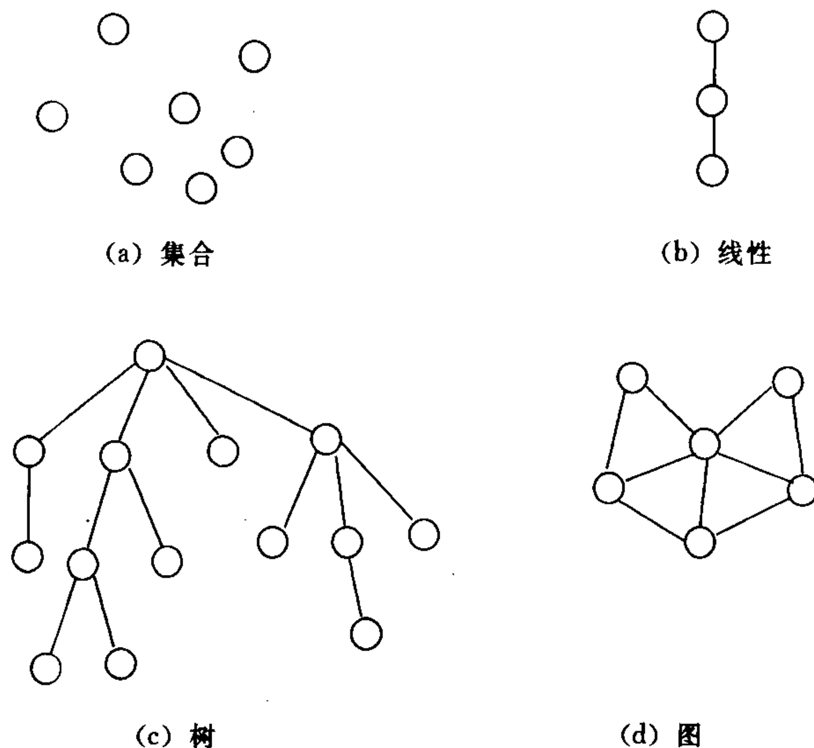


图 1-3　四类基本结构关系图

数据结构的形式定义可以用一个二元组表示:

Data－Structure＝(D,R)

其中,D 为具有相同特性的数据元素的有限集,R 是 D 上数据元素之间关系的有限集。例如:复数在计算机科学中可作如下定义:

Complex＝(C,R)

其中,$C=\{c_1,c_2 \mid c_1,c_2\in$ 实数$\}$,$R=\{\langle c_1,c_2\rangle \mid c_1$ 为实部,c_2 为虚部$\}$。又如:今欲编制一个企业管理程序,其中需要管理工厂中各车间的工人,则首先要为计算机处理的对象——车间工人设计一个数据结构,设一个车间由一个车间主任,三个班组的小组长以及 12 个工人组成,且这些成员之间的关系为:车间主任负责管理三个小组长,一

小组长又管理四个工人，则可以如下定义数据结构：

$$Grop=\{P,R\}$$

其中，$P=\{T,G_1,G_2,G_3,W_{11},W_{12},\cdots,W_{34}\}$

$$R=\{R_1,R_2\}$$

$$R_1=\{\langle T,Gi\rangle \mid 1\leqslant i\leqslant 3\}$$

$$R_2=\{\langle G_i,W_{ij}\rangle \mid 1\leqslant i\leqslant 3,1\leqslant j\leqslant 4\}$$

上述数据结构的定义仅是对操作对象的一种数学描述，而没有对操作对象定义具体的操作，换句话说，即从操作对象抽象出的数学模型。结构定义中的“关系”描述的是数据元素之间的逻辑关系，又称逻辑结构。

对数据结构的讨论，最终是为了在计算机中实现对数据元素的操作。为此我们不仅要讨论数据的逻辑结构及其运算，而且还要讨论数据结构在计算机中表示——数据的物理结构（又称存储结构），它包括数据元素的表示和元素之间关系的表示。数据元素之间关系的表示有两种不同的方法：顺序映象和非顺序映象。由此可得到两种不同的存储结构：顺序存储结构和链式存储结构（非顺序存储结构）。其中，顺序存储映象是借助元素在存储器中的相对位置来表示数据元素之间的逻辑关系；非顺序存储映象是借助在一个数据元素(a)中保存另一数据元素(b)在存储器中的相对位置来表示两个元素(a 和 b)之间的逻辑关系。

程序设计语言中，我们可以借助“数据类型”来描述数据的存储结构。例如：在 PASCAL 语言中，可以用“一维数组”描述顺序存储结构；用“指针”描述链式存储结构。

数据类型在程序设计语言中用以描述（程序）操作对象的特性。在高级语言中，每个变量都要属于一个确定的数据类型（整型、实型、字符型、布尔型等），不同的数据类型规定了在程序执行期间不同的变量的取值范围和允许的操作，所以，数据类型是一个值的集合和定义在这个值集上的一组操作的总称。例如，在 PASCAL 语言中的整数类型，它定义了其值集为区间[－maxin…maxin]上的整数（maxin 是依赖于具体计算机的最大整数），并规定了在这个值集上的一组操

作为：+、-、*、/、DIV、MOD 等。

与数据类型相关的还有一个概念称为数据对象。数据对象是某种数据元素的集合，是数据的一个子集。例如 PASCAL 语言中，布尔类型的数据对象是集合{TRUE(真)，FALSE(假)}，字符的数据对象为集合{x|x∈所有 ASCII 字符}。

二、数据结构的发展概况

六十年代初期，国内外的大学中都没有独立的“数据结构”课程，但数据结构的有关内容已散见于操作系统、编译原理和表处理语言等课程之中。1968 年，美国一些大学的计算机科学系的教学计划中明确规定“数据结构”为一门课程，但没有明确规定该课程的内容范围。当时，数据结构几乎和图论，特别是表和树的理论是同义语。随后，数据结构这个概念被扩充到包括网络、代数、集合论、关系等现在称之为“离散数学结构”的那些内容，它们同现在的“数据结构”的某些内容混在一起，总称为“数据结构”。由于数据必须在计算机中进行处理，因此，不能局限于研究数据本身的数学概念，还必须考虑数据的物理结构。这就进一步扩大了数据结构的内容。自从 1968 年美国研究计算机科学的著名教授 D. E. Knuth 所著的“计算机程序设计技巧”(The Art of Computer Programming)问世以后，才逐渐把数据的逻辑结构、物理结构以及每种结构所定义的运算作为组成“数据结构”课程的主要内容。近年来，由于数据库系统的不断发展，在数据结构课程中又增加了文件管理，特别是大型文件的组织等方面的内容。

数据结构与数学、计算机硬件、特别是计算机软件有着密切的关系。它是计算机专业的一门核心课程，是编译原理、操作系统、数据库、人工智能等课程的基础。同时又广泛用于信息科学、系统工程、应用数学以及各种工程技术领域。

数据结构在计算机科学中有着十分重要的地位。它有自己的理论、研究对象和应用范围，而且其研究的内容正在不断扩充和深化。而作为一门课程、一本教材，因受特定的对象和时期的限制，因此不能全面地反映数据结构的全貌。但值得注意的是，数据结构是新兴的

学科，正值尤兴未艾，蓬勃发展的阶段。一方面，面向各专门领域中特殊问题的数据结构得到研究和发展，如多维图形数据结构等；另一方面，从抽象数据类型的观点来讨论数据结构，已成为一种新的趋势，越来越被人们所重视。

习 题

1. 指出下列集合的元素是什么？
   (1) 我国四大发明的集合。
   (2) 小于 24 并且是 5 的倍数。
   (3) $B=\{w \mid |w|<3, w\in Z\}$。
   (4) 方程 $y^2-5y+4=0$ 在实数范围内的解。
2. 下列每一对集合是否相等？
   (1) $\{-1,1\}$, $\{1,-1\}$
   (2) $\{1,2,3\}$, $\{2,1,4\}$
   (3) $\{a,b,c\}$, $\{b,c,d,e,a\}$
   (4) $\{x \mid x^2-4x+3=0, x\in R\}$, $\{1,3\}$
3. 若集合 $A=\{a,b,c\}$，下面的记法哪些是正确的？哪些是不正确的？
   $a\in A$; $b\subset A$; $\{c\}\subset A$; $A\subset A$; $\Phi\subset A$; $\{c\}\in A$; $\Phi\in A$
4. 已知：$A=\{0,2,4,6,8\}$，$B=\{1,3,5,7,9\}$，$C=\{2,5,6,9\}$，求：
   $A\cap B$, $A\cap C$, $B\cap C$, $A\cup B$, $A\cup C$, $B\cup C$, A-B, B-C, C-A
5. 设 $A=\{2,3,4,6\}$、$B=\{a,b,c)$，求 $A\times B$ 和 $B\times A$。
6. 简述下列术语：数据、数据元素、数据对象、数据结构、存储结构和数据类型。
7. 数据结构研究的主要内容是什么？

第二日　算法的描述与分析

数据结构是一门实践性很强的学科,通过该课程的学习,读者能运用数据结构的技巧更好地进行算法和程序设计,所以我们在讨论各种数据结构的基本运算时,都给出了相应的算法。对于算法的描述,我们力求做到通俗易懂,适于自学,所以采用文字框图进行描述。读者在掌握和理解了框图所示的设计思想后,可以较方便地使用自己熟悉的算法语言来编制程序。另外,考虑用 PASCAL 语言来编写程序可以较好地体现程序的结构,并且简明易学,具有实用价值,所以本书中大部分算法在给出文字框图的同时还给出了用 PASCAL 语言编写的源程序片断。下面,我们将对本书框图中使用的符号及 PASCAL 语言进行介绍。

2.1 PASCAL 语言初步

一、程序结构

首先我们看一个 PASCAL 语言程序的例子,该程序要求:输入两个数,并且计算两数之和,然后再输出结果。

```
PROGRAM example(INPUT,OUTPUT);
  {This is used for s=x+y}
VAR x,y,s:ingeger;
BEGIN
  read (x,y);
  s:=x+y;
```

```
    writeln('s=',s)
END.
```

以上是一个用PASCAL语言编写的简单程序,它体现了PASCAL语言程序的结构由程序首部和分程序组成,而分程序又分为说明部分和语句部分。

1. 程序首部

程序首部用于指定程序的名字和列出程序中用到的文件。它由程序标识符、程序名、程序参数所组成。其中,程序标识符是PASCAL语言的标志,是每个PASCAL语言程序都有的,用PROGRAM表示。程序名事实上是一个标识符,是用户为程序安排的名字,就像一个人有一个人名一样,每个程序都有一个程序名为标记。如上例中的example就是一个程序名。在PASCAL语言中,标识符是一个字母开头的后跟若干字母或数字的字符串。例如:x,x12,x3yz等都是标识符,而123x就不是标识符。程序参数用来表示该程序同外界的联系。它们一般是文件名,最常用的文件为INPUT、OUTPUT,表示标准的输入/输出文件。

程序首部有时还带有程序的注释部分,用于对程序的名称、类型、功能、编写日期等进行描述。如上例中的{This is used for s=x+y}。

2. 程序的说明部分

在PASCAL程序中允许用户自己定义标号、常量、类型、变量、过程和函数等,这些都必须首先在程序的说明部分加以说明,然后才能在程序的执行部分引用。程序的说明部分应遵循如下次序:

(1)标号说明部分;

(2)常量定义部分;

(3)类型定义部分;

(4)变量说明部分;

(5)过程与函数说明部分。

下面予以逐一介绍。

■ 标号说明部分

在 PASCAL 语言中提供了 GOTO 语句。GOTO 语句使程序不再顺序地执行程序的下一个语句，而转去从指定标号的语句开始执行。标号规定为四位以内的无符号整数。标号和冒号“:”一起放在一个语句的前面，构成一个带标号的语句。例如：

```
99: Writeln('ERROR DETECTED');
```

GOTO 语句的例子为：GOTO 99

PASCAL 语言规定，语句前的标号以及 GOTO 语句的标号都必须在标号说明部分中说明。标号说明的一般形式为：

```
LABEL    标号表;
```

其中，标号表由一系列标号组成，标号之间用逗号隔开。例如：

```
LABEL 12 ,100;
```

说明了两个标号 12 和 100。

■ 常量定义部分

在程序中我们常会用到一些具体不变的数据，称为常量。例如，数学常数 π 的近似值 3.1415926。在程序中允许在哪里用到一个常量，就在那里写出这个常量的值。但是，为了程序描述清晰，书写方便，修改容易，程序员宁愿在用到常量的地方使用一个标识符代替常量的值。常量定义就是用来引入一个标识符作为一个常量的同义词。凡是在程序中出现这个标识符，就等价于在那里直接写上相应的常量值。如果要修改某一常量，只要修改一下它的常量定义即可。常量定义的一般形式为：

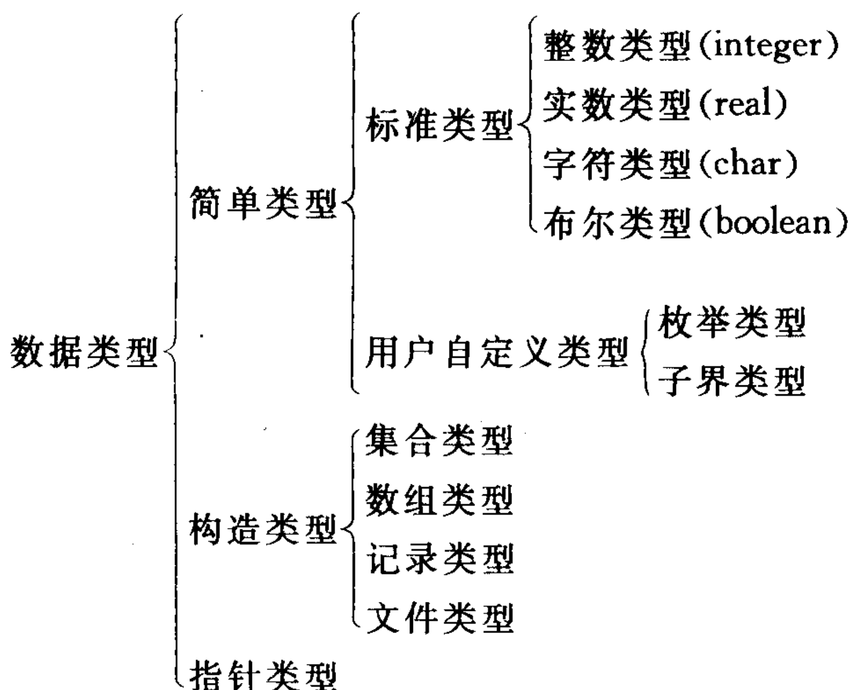
```
CONST      标识符 1=常量值 1;
           标识符 2=常量值 2;
                  ⋮
           标识符 n=常量值 n;
```

下面是一个常量定义的实例，它定义了三个常量：Pi、Epsilon、Max：

```
CONST      Pi=3.1415926 ;
           Epsilon=1E-6 ;
           Max=100;
```

■ 类型定义部分

PASCAL 语言的数据类型可以图示如下：



对四种标准类型，程序员不用预先定义就可以引用。此外，程序员也能根据需要自己定义类型，并给一个类型标识符。类型定义的一般形式为：

```
TYPE  类型标识符 1=类型 1;
      类型标识符 2=类型 2;
              ⋮
      类型标识符 n=类型 n;
```

其中，类型标识符是指所定义类型的名称，类型指已定义的各类型，具体定义后面介绍。

■ 变量说明部分

在程序中除了常量外还经常会用到另外一种数据，它的值在程序执行期间可以根据需要而变化，我们称之为变量。每个变量必须有一个标识符，并属于某一类型。变量说明就是把变量标识符和它的类型联系起来。变量说明的形式为：

```
VAR  标识符表 1:类型名 1;
     标识符表 2:类型名 2;
```

⋮

标识符表 n:类型名 n;

其中,标识符表由一个或多个标识符组成,标识符之间用逗号分隔。

例如:

```
VAR i,index:integer;
    root1,root2:real;
    ch:char;
    found:boolean;
```

在这个例子中,把 i,index 说明成整型变量;root1,root2 说明成实型变量;ch 说明成字符型变量;found 说明成布尔型变量。

■ 过程和函数定义部分

在 PASCAL 语言中,不严格地说,过程(或函数)说明是定义了逻辑上相关的语句序列,用于描述一组复合的动作。并且给这个语句序列取一个名字,即过程(或函数)名。过程(或函数)定义的结构和程序类似。

如下是一个过程定义的例子:

```
PROCEDURE add(x,y:integer; VAR sum:integer);
BEGIN
  sum:=x+y;
END;
```

它体现了 PASCAL 语言中过程定义的结构由过程首部和分程序组成。其中分程序的定义和程序中的分程序一样分为说明部分和语句部分。过程首部用于指定过程的名字和列出过程中用到的参数,它由过程标识符、过程名、形式参数所组成。其中,过程标识符为 PROCEDURE,是每个过程都有的;过程名是一个标识符,是用户为过程起的名字,上例中的 add 就是一个过程名;形式参数用来表示该过程同外界的联系,上例中定义了三个参数 x、y 和 sum ,它们都为整型数。其中,x 和 y 称为值参,它只能从外界把数据送进来,而不能把数据送出去;sum 称为变参,它不仅可从外界把数据送进来,而且通过它能把数据送出去。

下面是一个函数定义的例子：

```
FUNCTION eq(x,y:integer):boolean;
BEGIN
  IF x=y then return TRUE
          ELSE return FALSE
END;
```

函数的结构和过程类似也由函数首部和分程序组成。其中分程序的定义和程序中的分程序一样又分为说明部分和语句部分。函数首部同过程首部不同之处在于函数的标识符是 FUNCTION；并且函数需要指明返回值的类型。上例中函数 eq 返回值为布尔类型。在函数中要使用 return 语句返回函数值。

二、PASCAL 语言的语句

PASCAL 程序的语句部分由如下形式定义：

```
BEGIN
  语句 1;
  语句 2;
    ⋮
  语句 n;
END
```

其中，语句是指如下所述的执行性语句。

1. 赋值语句

一般形式：

变量：=表达式：

其作用就是计算右端的表达式，并把结果赋给左端的变量。其中表达式可以是算术表达式、关系表达式和布尔表达式。

PASCAL 语言的算术表达式是整型量或实型量与算术运算符的合法组合。其中算术运算符为：

+(加)、-(减)、*(乘)、DIV(整除)、MOD(求余)、/(实数除)。

它的规定如下：

(1) ＋(加)、－(减)、*(乘)是三种常用的运算,当两个操作数都是整型数时,运算结果是整型数;其中,若有一个实数或两个都是实数,则结果为实数。

(2) /(实数除)不管操作数是实数还是整数,结果均为实数。DIV(整数除)和 MOD(求余)运算的两个操作数必须都是整型数,结果也是整数。

PASCAL 语言的关系表达式是算术表达式与关系运算符的合法组合。关系运算符主要有下列六种:

＝(等于),〈 〉(不等于),＜(小于),＞(大于),＜＝(小于等于),＞＝(大于等于)关系运算的结果为布尔型数值:真(TRUE)值和假(FALSE)值。一个关系表达式实际上表示了一个判定条件,若判定条件满足,则关系表达式的结果为 TRUE,否则为 FALSE。

PASCLA 语言的布尔表达式是由布尔型数据和布尔(逻辑)运算符组成。其中布尔型数据可以由产生布尔型数值的关系表达式代替;布尔运算符主要有下列三种:NOT、AND 以及 OR 运算。其运算法则为:

A	B	A AND B	A OR B	NOT A
TRUE	TRUE	TRUE	TRUE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE	FALSE
FALSE	FALSE	FALSE	FALSE	TRUE

需要注意的是:

(1) 数学上的表达式:A≥B≥C≥D,在 PASCAL 语言必须写成:

(A＞＝B) AND (B＞＝C) AND (C＞＝D)

(2) 逻辑运算必须写成诸如:

A AND B,A OR B,NOT A;

(3) 运算优先次序为:

① 圆括号,由内向外逐层展开;

② NOT ;

③ AND ;

④ OR ;

⑤ * 、/、DIV、MOD;

⑥ +、-;

⑦ =、< >、>、<、>=、<=;

⑧ 同一级从左到右顺序计算。

2. 复合语句

一般形式:

```
BEGIN
    语句 1;
    语句 2;
      ⋮
    语句 n
END
```

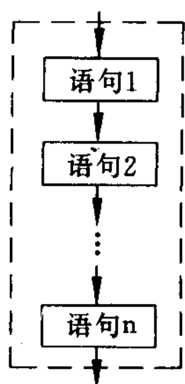


图 2-1 复合语句

其中,语句是指 PASCAL 语言中的任何语句。

复合语句是由一对 BEGIN 和 END 括起来的语句序列所组成。BEGIN 和 END 起着语句括号的作用;其中的语句是构成复合语句的成分,称为成分语句,它们用分号分隔。复合语句的执行,就是按程序正文的书写顺序执行语句括号里的成分语句(除非 GOTO 语句才能改变执行成分语句的顺序),其执行过程如图 2-1 所示。整个虚框看作是一个语句,它有一个入口和一个出口。

3. 如果(IF)语句

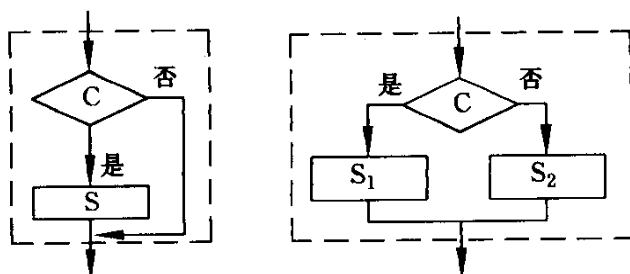
一般形式:

IF　布尔表达式　THEN　语句 1

或　IF　布尔表达式　THEN　语句 1　ELSE　语句 2

其中,语句为 PASCAL 语言中的任何语句。

如果语句有两种形式:第一种称为 IF-THEN 语句,当布尔表达式的值为 TRUE,则执行 THEN 后面的语句;否则,不执行(这时



(a) IF C THEN S　(b) IF C THEN S_1 ELSE S_2

图 2-2　如果语句

的如果语句等效于空语句)。第二种称为 IF－THEN－ELSE 语句，当布尔表达式的值为 TRUE 时，则执行 THEN 后面的语句 1；否则执行 ELSE 后面的语句 2。它们的执行过程分别如图 2-2(a)、(b)所示，其中，c 表示布尔表达式，s、s_1、s_2 分别表示语句、语句 1 和语句 2。

在如果语句中，若语句 1 或语句 2 又是一个如果语句，则称为嵌套的 IF 语句。

例如：
```
IF 条件 1
    THEN IF 条件 2
             THEN 语句 1
             ELSE  语句 2
    ELSE 语句 3 ；
```

对于嵌套的 IF 语句，有可能产生二义性。例如：

IF c_1 THEN IF c_2 THEN s_1 ELSE s_2

其中 ELSE 可以同第一个 THEN 配对：

IF c_1 THEN IF c_2 THEN s_1
ELSE s_2

也可以同第二个 THEN 配对：

IF c_1 THEN
IF c_2 THEN s_1
ELSE s_2

在 PASCAL 语言中为解决这一问题，规定 ELSE 是与它前面最邻近的还没有配对的 THEN 配对的。即在上例中 ELSE 规定同第二个

THEN 配对。

如果在程序中需要 ELSE 和第一个 THEN 配对，则可以对嵌套的内层 IF 语句加一对 BEGIN 和 END：

IF c_1 THEN BEGIN IF c_2 THEN s_1 END

ELSE s_2

4. 分情形(CASE)语句

一般形式：

```
CASE 表达式 OF
  分情形表 1 :  语句 1;
  分情形表 2 :  语句 2;
             ⋮
  分情形表 n :  语句 n;
END
```

其中，分情形表由一个或多个常量组成，常量之间由逗号分隔，语句为 PASCAL 语言中的任何语句。

在 CASE 语句中，分情形表内的常量必须和表达式的类型相同，且必须是序数类型(整型、字符型和枚举型)。表达式起着选择器的作用，当表达式的值等于某一常量时，则执行这个标号后指明的语句，这个语句执行完毕，分情形语句就完成了。如果表达式的值与任一常量都不符合，则 CASE 语句相当于一个空语句。分情形语句的执行过程如图 2-3 所示。

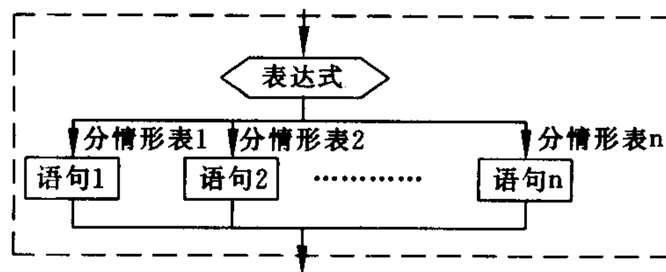


图 2-3 分情形语句

5. 当(WHILE)语句

一般形式：

WHILE 布尔表达式 DO 语句

其中，语句为 PASCAL 语言中的任何语句，它体现了循环的动作，所以又称为 WHILE 语句的循环体。

WHILE 语句的执行过程为：

(1) 计算布尔表达式的值，若结果为 TRUE，则转(2)；否则，转(3)；

(2) 执行 WHILE 循环体中的语句，然后转(1)；

(3) WHILE 语句执行结束。

其框图如图 2-4 所示。

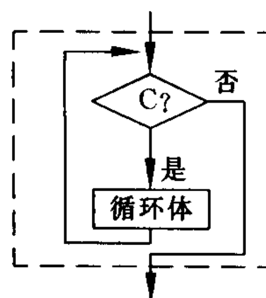


图 2-4 当语句

6. 重复(REPEAT)语句

一般形式：

REPEAT

　　语句 1；

　　语句 2；

　　　⋮

　　语句 n

UNTIL 布尔表达式

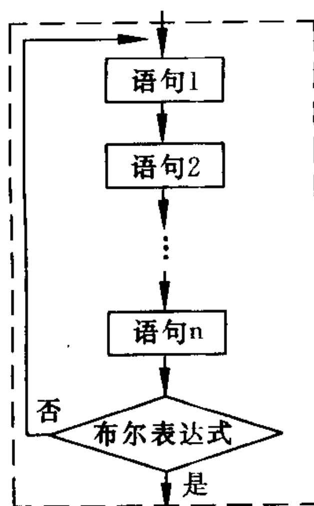


图 2-5 重复语句

其中，语句序列组成了 REPEAT 语句的成分语句，称为 REPEAT 语句的循环体。这里 REPEAT 和 UNTIL 起着一对语句括号的作用。布尔表达式是结束条件，其执行过程如下：

(1) 执行循环体(语句 1；语句 2；...；语句 n)；

(2) 计算布尔表达式，结果为 FALSE 转(1)；否则转(3)；

(3) 重复结束。

其框图如图 2-5 所示。

7. 循环(FOR)语句

一般形式：

FOR 循环变量：=初值表达式 TO 终值表达式 DO 语句

或 FOR 循环变量：=初值表达式 DOWNTO 终值表达式 DO 语句

其中，循环变量必须是序数类型，它与初值、终值两个表达式的类型必须赋值相容。根据从初值到终值是升序的还是降序的，FOR 语句分别有 TO 和 DOWNTO 两种形式。其中，前者要求初值表达式的值≤终值表达式的值；后者要求初值表达式的值≥终值表达式的值。

FOR 语句的执行过程为：

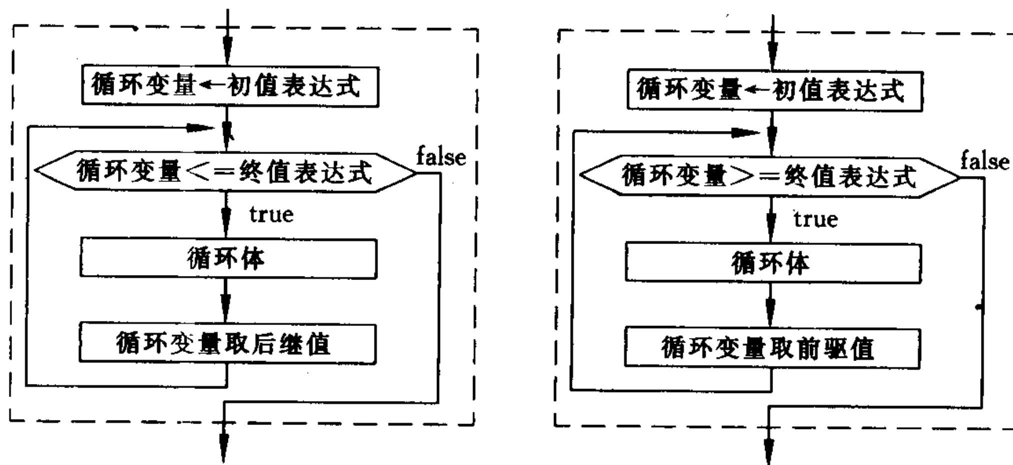
(1) 计算初值表达式和终值表达式，并把初值表达式的值赋给循环变量；

(2) 循环变量与终值表达式的值进行比较。对于 TO 形式，当循环变量≤终值表达式时转(3)；否则转(4)。对于 DOWNTO 形式，当循环变量≥终值表达式时转(3)；否则，转(4)；

(3) 首先执行循环体。然后，对于 TO 形式，循环变量取它的后继值；对于 DOWNTO 形式，循环变量取它的前驱值。转(2)；

(4) 循环结束。

图 2-6 为 FOR 语句的框图：



(a) TO 形式　　(b) DOWN TO 形式

图 2-6　循环语句

8. 转移(GOTO)语句

一般形式:

GOTO 语句标号

其中,语句标号是在标号说明部分说明的一个带标号的语句的标号。

GOTO 语句不再顺序地执行程序,而是转去执行指定标号的语句,由于 GOTO 语句破坏了程序的结构,一般不提倡使用。

9. 过程语句

一般形式:

过程名(实参表)

或 过程名

其中,过程名是在过程定义中定义的过程名。当在过程定义中有参数说明(形式参数表)时,过程语句必须用第一种形式,在过程调用时写上同形式参数相匹配的实际参数。过程语句的执行等效于执行相应过程定义所包含的语句序列。

在 PASCAL 语言中提供了一些标准的输入/输出过程:read(参量表),readln(参量表),write(参量表),writeln(参量表)。

其中:

(1) read,readln 用于输入数据;write,writeln 用于输出数据。

(2) read 要求至少输入一个数据,但不要求换行;readln 要求输入完数据后换行,但可以不输入任何数据,只执行换行。

(3) write 要求至少输出一个数据,但数据输出完不换行;writeln 数据输出后换行,但可以不输出任何数据,只执行换行。

三、用户自定义的数据类型

1. 枚举类型

一般形式:

TYPE 类型标识符=(标识符 1,标识符 2,…,标识符 n);

枚举类型的值是一些标识符,称为枚举标识符。枚举类型是通过列举所有的枚举标识符定义一个有序集合。枚举类型的数据只能取这些标识符中的一个。枚举类型是有序的,每个枚举标识符都有一个

序数，第一个枚举标识符的序数为 0，此后，按枚举的次序，依次为 1，2，…。

例如：用枚举类型定义一个星期

TYPE day=(Sun,Mon,Tues,Wed,Thur,Fri,Sat);

对于枚举类型的数据有以下标准函数：succ(x)(求 x 的后继)，pred(x)(求 x 的前趋)，ord(x)(求 x 的序号)。例如：

succ(Mon)=Tues，pre(Mon)=Sun，ord(Mon)=1

2. 子界类型

一般形式：

TYPE　类型标识符=常量 1...常量 2;

其中，常量 1，常量 2 分别称为下界和上界，必须属于同一类型，一般为整型、字符型、枚举型，而且 ord(常量 1)<ord(常量 2)。

子界类型定义了一个已定义过的类型的子界，该已定义过的类型称为主类型。一个子界类型的数据和它的主类型数据具有相同的运算操作，只是其取值范围只能在子界之内。

例如：把星期一到星期六定义成工作日，它是类型 day 的一个子界类型。

TYPE workday=(Mon..Sat);

3. 集合类型

一般形式：

TYPE 类型标识符=SET OF 基类型;

其中，基类型只能是序数类型(整型，字符型，枚举型，子界型，布尔型)。集合类型的数据是基类型数据值的集合的子集，对于集合类型数据可以作关系运算和集合运算。关系运算有五种：

(1) = ：判两个集合是否相等，相等时结果为 TRUE；否则为 FALSE；

(2) <>：判两个集合不等，不相等时结果为 TRUE；否则为 FALSE；

(3) <=：判左边的集合是否蕴含于右边的集合，结果为布尔值；

(4) >=：判左边的集合是否蕴含右边的集合，结果为布尔值；

(5) IN：判一个元素是否属于一个集合。若集合 A 中存在元素 x，则 x IN A 的结果为 TRUE；否则为 FALSE；

集合运算有三种：集合并(+)，集合交(*)，集合差(-)。

下面是一个集合定义的例子，它定义十个数字字符的集合：

TYPE digitset=SET OF '0'…'9'；

4. 数组类型

一般形式：

TYPE 类型标识符=ARRAY[下标类型 1，…，下标类型 n] OF 成分类型。

其中，下标类型可以有一个或多个，它只能是枚举型，子界型，字符型之一。数组有一个下标时，称为一维数组；有 n 个下标时，称为 n 维数组。成分类型可以是 PASCAL 语言中的任何类型，它规定了数组成分的类型。

数组类型是由固定数量的具有相同类型的成分变量组成。每个成分变量是通过数组变量名跟以数组下标来直接访问的；下标可以是表达式，其类型同相应的下标类型一致；这种访问是完全随机的，故数组被称为随机访问结构。

例如：定义了 30 个学生的成绩为一个数组，

TYPE studentgrade=ARRAY[1..30] OF real；

这是一个一维数组，它的下标类型是整型的子界 1..30，其成分类型为实型，studentgrade 是数组类型标识符。定义了一个数组之后，在变量说明中，就可以按通常方式说明数组变量：

VAR student ：studentgrade ；

数组变量 student 有 30 个成分变量：student[1]，student[2]，…，student[30]，每个成分变量都是实型变量。

5. 记录类型

一般形式：

TYPE 类型标识符 = RECORD 域表 end；

域表 = 固定部分

或 固定部分;变体部分

或 变体部分

其中,固定部分为:记录项 1;记录项 2;…;记录项 n。

记录项定义为:项标识符 1,项标识符 2,…,项标识符 n:类型

从记录的固定部分可以看出,一个记录由一些记录项组成,每个记录项含有若干个项标识符,并且对它规定类型。

例如:定义由年、月、日三项组成的日期这样的数据类型如下:

```
TYPE mon=(Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct ,Nov,
Dec);
    date=RECORD
           year : integer;
           month : mon;
           day : 1…31
         END;
  VAR today : date;
```

在这里,year、month、day 称为项,today 被说明成 date 型的变量。对于记录变量可以访问其某一个成份(项)。记录变量的成份表示为:记录变量名.成分名;上例中为 today.year,today.month,today.day。

在一小段程序中如需多次访问一个记录的同一成分或同一记录的不同成分,可以使用开域语句 WITH 以提供一种缩写表示法。开域语句的形式为:

WITH 记录变量 DO 语句

在语句中只要直接使用该记录的记录项即可。

例:

```
WITH today DO
  BEGIN
    year:=1994;
    month:=Jan;
    day:=10;
  END;
```

在这里就不用写成：today. year：=1994；…。

记录类型的语法还提供了变体部分，它允许同一种记录类型的变量具有一些不同的结构，表现在它们中某些成分的数目和类型上的不同。例如：我们定义一个人的记录包括：人名、生日、性别和婚姻状况，其中在婚姻状况中，对结婚的人记录配偶的姓名和结婚日期；对离过婚的人记录离婚日期和是否第一次离婚；对单身的人记录是否独居。其形式如下：

```
TYPE alfa=ARRAY[1…20] OF char ;
     date= {如上}
     marstatus=(married,divorced,single);
     person=RECORD
              name : alfa;
              birthday : date;
              sex : (male,female);
              CASE ms :marstatus OF
                married : (spousename:alfa; mdate:date);
                divorced : (ddate:date; fstd:boolean);
                single : (indepdt:boolean)
              END;
```

6. 指针类型

一般形式：

TYPE 类型标识符=↑目标类型；

其中，目标类型是 PASCAL 语言中的任何类型标识符。我们称定义的类型为指向目标类型数据的指针类型。

例：

```
TYPE link=↑node ;
     node=RECORD
            data : integer ;
            next : link
          END;
VAR p : link ;
```

在这里我们定义了 link 为指向 node 类型数据的指针类型。指针

类型是一种动态的数据类型，一旦程序执行时需要生成一个 node 类型的新变量（动态变量）时，可以调用标准过程：

new(p)；

其中，p 为指针变量，作为过程 new 的实在参数。调用该过程将动态分配一个 node 型变量的存储空间，产生指向这个新变量的存储地址，并把该地址作为指针值赋给指针变量 p。由于新变量并不经变量说明，所以不采用象静态变量那样的标识符，而是用 p↑来标记它，p↑称为引用变量。必须千万注意，不要混淆指针 P（属于指针型 link 或↑node）和它所指向的变量 p↑（属于类型 node）。如图 2-7 为指针与它所指变量的示意：

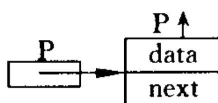


图 2-7　指针 p 和它所指向的变量 p↑

当程序执行时需要取消一个由 new 产生的动态变量时，可以调用如下标准过程：

dispose(p)；

其中，P 为指针变量，它作为过程 dispose 的实在参数。调用该过程将 p 指向的动态变量 p↑的存储空间释放，使 p↑变为不可访问，p 的值也就无定义。

2.2　框图介绍和算法分析

在上一节我们简单地介绍了有关 PASCAL 语言的内容。本书中算法主要是以文字、框图的形式进行描述。下面对框图中使用的符号分别介绍。

本书框图中使用的图形符号如图 2-8 所示。

1. 椭圆框：框中用文字标明算法的“开始”或“结束”。

2. 矩形框：框内描述某些操作，如赋值、组织循环、输入和输出等，统称为操作框。

3. 菱形框（包括变相菱形框）：称为判别框，框中符号冒号“:”表示比较。例如 n:0 表示 n 与 0 相比较，比较的结果写于框外连接线条的旁边．带箭头的线条表示算法或程序的走向，写于其旁的判断

结果就是算法或程序的分支走向应满足的条件。

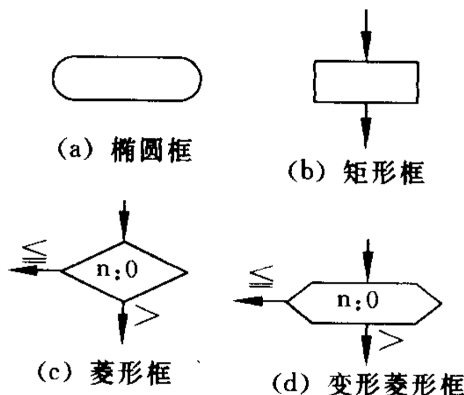


图 2-8 框图所用图形

对算法(或程序)的分析和评价通常较复杂,一般需考虑正确性、简单性、最优性、运算量及占用存储量等诸多因素。为了简化讨论,我们仅采用其中的两条标准:其一是用问题的某个参数的函数来估算其存储量;其二是讨论数据运算所需的计算量。假设问题的参数为 n,此参数可以是矩阵的阶、线性表的长度、图的顶点数等显示该问题规模大小的参数,那么在所选数据结构上执行有关操作所需要的存储量及计算量是 n 的什么函数呢?我们引进记号"O",对这些函数作数量级的估算。例如,对于下述三个简单程序段:

(1) x:=x=1;

(2) FOR i:=1 TO n DO x:=x=1;

(3) FOR i:=1 TO n DO

    FOR j:=1 TO n DO x:=x+1;

在程序段中 1 中,语句 x:=x+1 不包含在任何循环体之中,此语句只执行一次,计算量可记为 O(1)。在程序段 2 中,上述赋值语句在 FOR 循环之中,所以要执行 n 次,其执行时间和 n 成正比,计算量可记为 O(n)。在程序段 3 中,x:=x+1 语句要执行 n×n 次,其执行时间和 n^2 成正比,故计算量可记为 $O(n^2)$。对于算法或程序所需的存储量,也可以作类似分析。然而,要事先对一个算法的计算量作仔细的分析很复杂,而且也不是本课程的主要内容,所以书中对一些算法的性能评价只根据算法中执行次数最多的语句来估算其计算量的数量级。

习　题

1. 下面程序的执行结果是什么？

```
PRAGRAM ex(input,output);
CONST n=4;
VAR x,p,i,sum:integer;
BEGIN
  sum:=0;
  FOR x:=1 TO 4 DO
    BEGIN
      p:=1;
      FOR i:=1 TO x DO
          p:=p * x;
      sum:=sum+p
    END;
  writeln(sum)
END
```

2. 请用框图（流程图）描述下列问题的算法。

   (1) 输入三个数 a、b、c，要求按从小到大的次序输出。

   (2) 找出一组数 a[1],a[2],…,a[n]中的最大数和最小数。

3. 找出下面 PASCAL 语言程序中的语法错误。

```
PRAGRM   ex$w(input,output)
CONSt    pi:=3.14159;
         bool=false;
TYPE     k=1.0..10.0;
VAR      i,j: integer;
         bool:boolean;
         p:node;
BEGIN
  sum:= 0
  FOR I:=1 TO 100
    sum:=sum+i;
  write("sum=",sum)
END
```

第三日　线性表

从第三日到第六日我们将讨论线性结构:线性表、栈、队列、串、数组。线性结构的特点是:在数据元素的非空有限集合中,存在唯一的一个数据元素被称为第一个元素;存在唯一的一个数据元素被称为最后一个元素;除第一个元素之外,其余的数据元素均有唯一的直接前驱元素;除最后一个元素之外,其余的数据元素均有唯一的直接后继元素。在以后各日讨论各种数据结构时,我们都将首先介绍有关结构的基本概念和基本运算,然后讨论它在计算机中的存储结构以及在不同的存储结构上相应操作的实现,最后再介绍有关应用。下面我们将按上述顺序首先介绍线性表。

3.1　线性表及其基本操作

一、线性表的基本概念

线性表(linear-list)是最常用、最简单的一种数据结构。简而言之,一个线性表是($n \geq 0$)个数据元素($a_1, a_2, \cdots, a_n$)的有限序列。数据元素的具体含义是各种各样的,可以是一个数,一个字符,或一张表格等。以下是三个线性表的例子。

例 1:(70,32,51,101,6)是一个线性表,其中的数据元素为整数,共有 5 个数据元素。

例 2:(A,B,C,…,Z)是一个线性表,其中的数据元素为大写英文字母,共有 26 个数据元素。

例 3:图 3-1 所示的学生一门课程的成绩单也是一个线性表。

姓名	学号	性别	成绩
陈哲宇	8962101	男	89
吴斐	8962102	女	91
翁雷	8962103	男	76
龚梅	8962104	女	78
施咏	8962105	男	92
⋮	⋮	⋮	⋮

图 3-1 学生成绩单

其中的数据元素是每一个学生所占据的整栏表格，包括姓名、学号、性别、成绩共有四个数据项。通常把这种由多个数据项组成的一个数据元素称为一个记录。

综合上述三例，可以对线性表作如下的形式定义：

含有 n 个数据元素的线性表是一个数据结构：

linear-list＝(D,R)

其中，$D=\{a_i|a_i\in datatype,\ 1\leqslant i\leqslant n,n\geqslant 0\}$

$R=\{N\},N=\{\langle a_{i-1},a_i\rangle|a_{i-1},a_{i1}\in datatype,2\leqslant i\leqslant n\}$

datatype 为某种数据对象

从定义中我们可以得知：不同线性表中的数据元素可以是名种各样的，但同一线性表中的元素必定具有相同的性质，属于同一数据对象。关系 R 是一个序偶的集合，它表示线性表中数据元素之间的相邻关系，即 a_{i-1}领先于 a_i，a_i 领先于 a_{i+1}。我们称 a_{i-1}是 a_i 的直接前驱元素；称 a_i 是 a_{i-1}的直接后继元素。线性表中数据元素的个数 n 称为线性表的长度。当 n＝0 时，称为空表；当 n＞0 时，线性表通常记为 $(a_1,a_2,\cdots,a_i,\cdots,a_n)$，其中 a_1 称为第一个数据元素，a_n 是最后一个数据元素。当 $i=1,2,\cdots,n-1$ 时，a_i 有且仅有一个直接后继 a_{i+1}；当 $i=2,3,\cdots,n$ 时，a_i 有且仅有一个直接前驱 a_{i-1}。因此，线性表是一个线性结构。

二、对线性表的操作

线性表是一种非常灵活的数据结构，对线性表中的数据元素不仅可以进行访问，还可以进行插入和删除操作。对线性表的基本操作有以下几种：

1. 初始化(initial(L))：设定一个空的线性表 L。

2. 求长度(length(L))：运算的结果是求得线性表的长度，即给

出线性表 L 中数据元素的个数。

3. 存取元素(get(L,i)):此运算仅当 $1 \leqslant i \leqslant length(L)$ 时有意义,其结果为取线性表 L 中第 i 个数据元素。

4. 定位(locate(L,x)):若线性表 L 中存在值为 x 的数据元素 a_i,则运算结果为 a_i 在线性表中的序号 i;否则为零。若存在多个值为 x 的数据元素,则结果为其中序号最小的一个。

5. 插入(insert(L,i,b)):把给定元素 b 插入到线性表 L 的第 i 个位置,使长度为 n 的线性表 $(a_1,a_2,\cdots,a_{i-1},a_i,\cdots,a_n)$,变成长度为 n+1 的线性表 $(a_1,a_2,\cdots,a_{i-1},b,a_i,\cdots,a_n)$。

6. 删除(delete(L,i,b)):此运算仅当 $1 \leqslant i \leqslant length(L)$ 时才有意义,结果就是把线性表 L 的第 i 个数据元素 a_i 从 L 中删除,使长度为 n 的线性表 $(a_1,a_2,\cdots,a_{i-1},a_i,a_{i+1},\cdots,a_n)$ 变成长度为 n-1 的线性表 $(a_i,\cdots,a_{i-1},a_{i+1},\cdots,a_n)$。

7. 判空表(empty(L)):若 L 为空表,则返回布尔值 TRUE;否则,返回布尔值 FALSE。

除上述基本运算外,对线性表还可以进行一些更复杂的运算。如:将两个或两个以上的线性表合并成一个线性表;把一个线性表分解成两个或两个以上的线性表;重新复制一个线性表;对线性表中的数据元素按某个数据项递增(或递减)的顺序进行重新排列等等。这些操作均可利用上述的基本运算来实现。

3.2 线性表的顺序存储结构

在计算机的内存中,可以用不同的方式来存储一个线性表。我们将介绍两种基本的方法,即顺序存储和链式存储。在这一小节中讨论顺序存储。

一、顺序存储

计算机的内存是由有限多个存储单元组成的,每个存储单元都有唯一的地址,各存储单元的地址是连续编号的。对于一个线性表,

如果用一组连续的存储单元依次存放它的各个数据元素，这就是线性表的顺序存储。

假设线性表的每个元素需占 C 个单元，并且每个元素用其所占的第一个存储单元的地址作为该数据元素的存储位置，则线性表中第 i+1 个数据元素的存储位置 $Loc(a_{i+1})$ 与第 i 个元素的存储位置 $Loc(a_i)$ 之间满足下列关系：

$$Loc(a_{i+1})=loc(a_i)+c$$

一般来说，线性表中第 i 个元素的存储位置为：

$$Loc(a_i)=Loc(a_1)+(i-1)*c$$

其中，$Loc(a_1)$ 是线性表中第一个数据元素 a_1 的存储位置，通常称它为线性表的起始位置或基地址。如图 3-2 是线性表顺序存储的示意图：

存储位置	内存状态	元素序号
L	a_1	1
L+C	a_2	2
	⋮	⋮
L+(n-1)*C	a_n	n

图 3-2　线性表顺序存储结构示意图

线性表顺序存储结构的特点是为表中逻辑上相邻的元素 a_i 和 a_{i+1} 赋以相邻的存储位置 $Loc(a_i)$ 和 $Loc(a_{i+1})$。也就是说，以元素在计算机中存储位置的相邻来表示线性表中数据元素之间逻辑上的相邻关系。每一个数据元素的存储位置和线性表的起始位置相差一个和数据元素在线性表中的序号成正比的常数。所以，只要确定了起始位置，线性表中任一数据元素均可随机存取，因此线性表的顺序存储结构是一种随机存取的存储结构。在 PASCAL 语言中可以用一维数组（向量）来描述线性表的顺序存储结构，其描述如下：

```
CONST maxlength={线性表可能的最大长度}；
TYPE sequenlist=RECORD
        elements:ARRAY[1…maxlength] OF elementtype;
        last  :integer;
        END;
```

在上述描述中，线性表顺序存储结构是一个记录型的结构，其

中数据域 elements 描述了线性表中数据元素占用的向量空间，向量的第 i 个分量为线性表中第 i 个数据元素的存储映象，每个数据元素的类型为 elementtype；数据域 last 指示最后一个数据元素在向量空间中的位置。在此，数据元素的存储位置可以用其在向量中的下标值来表示。

在这种存储结构中，线性表的某些运算很容易实现。如：线性表的长度即为 last 域的值等等。下面讨论在这种存储结构下，线性表的插入和删除两种运算。

二、线性表顺序存储结构下的插入

线性表的插入操作是指在线性表的第 i－1 个数据元素和第 i 个数据元素之间插入一个新的数据元素 b。这就需要将第 i 个到第 n 个数据元素均向后移动一个位置，然后将新元素 b 存入向量的第 i 个位置。图 3-3 是线性表插入操作的示意图：

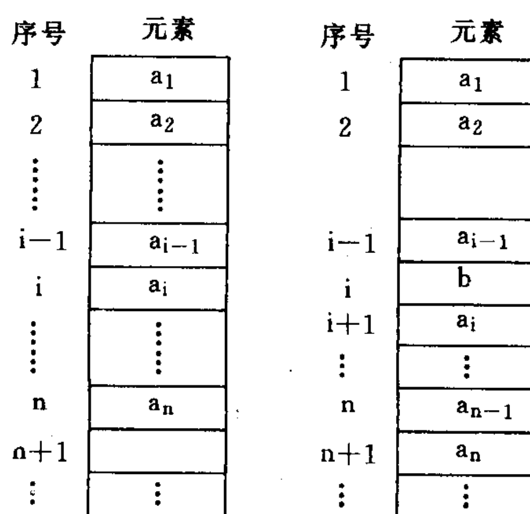


图 3-3　线性表插入示意图

线性表插入新元素 b 后，仍是一个线性表，不同的是其长度由原来的 n 变为 n＋1，数据元素 a_{i-1}和 a_i 之间的逻辑关系发生了变化，而其存储结构还是顺序存储。

当我们将插入操作写成算法时，还要考虑插入算法的通用性和可能出现的错误。例如，当给定的线性表已占满整个向量空间时，再

插入新元素就要溢出或插入位置不在线性表的范围内时就要出错。图 3-4 是在顺序存储结构下插入算法的框图描述。

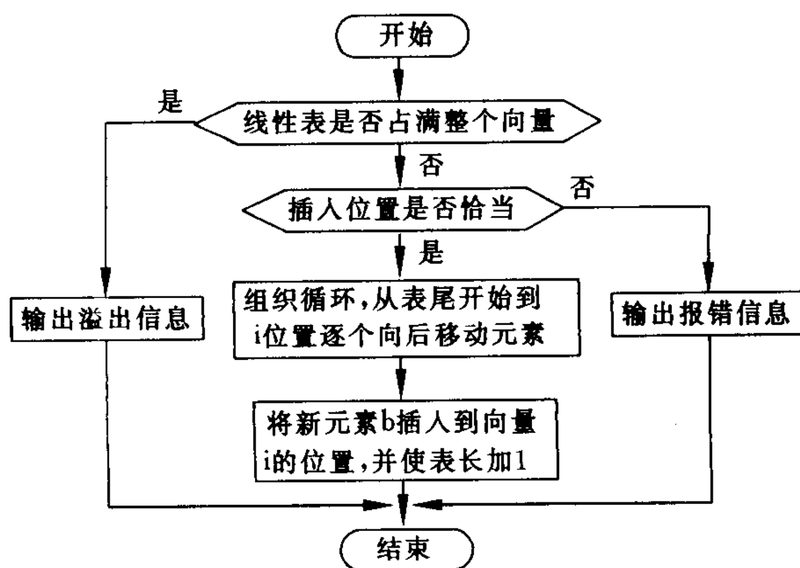


图 3-4　线性表顺序存储结构的插入算法框图

在线性表的顺序存储结构中,插入元素平均移动元素的次数是比较多的。如果在第一个元素之前插入新元素则要将线性表中所有的元素全部向后移动一个位置;只有在表尾插入一个新元素时,才不需要移动数据元素。假设在线性表的任何位置(包括在表尾)插入新元素的概率是相等的,即在第 i(i=1,…,n+1)位置上插入新元素的概率都等于 1/(n+1),则插入操作中元素平均移动次数为

$$E_{in}=(n+(n-1)+...+1+0)/(n+1)$$
$$=n/2$$

三、线性表顺序存储结构的删除

线性表的删除操作是把线性表中的第 i 个元素删去。这就需要将第 i+1 个到第 n 个元素向前移一个位置,并且线性表的长度减 1,删除算法的框图描述如图 3-5 所示。

对于线性表顺序存储结构的删除操作,假定线性表的长度为 n,且删除线性表中任何位置上的数据元素的概率相等,即等于 1/n,则删除操作中平均的元素移动次数为

$$E_{de}=((n-1)+(n-2)+...+1+0)/n=(n-1)/2$$

由上述操作可知，对于线性表的顺序存储结构，其结构简单且便于随机访问线性表中的任一元素，但不便于进行插入和删除操作；另外，如果要扩大向量的容量往往会很困难。为了克服顺序存储结构的缺点，人们提出了另外一种存储结构——链式存储结构(链表)。

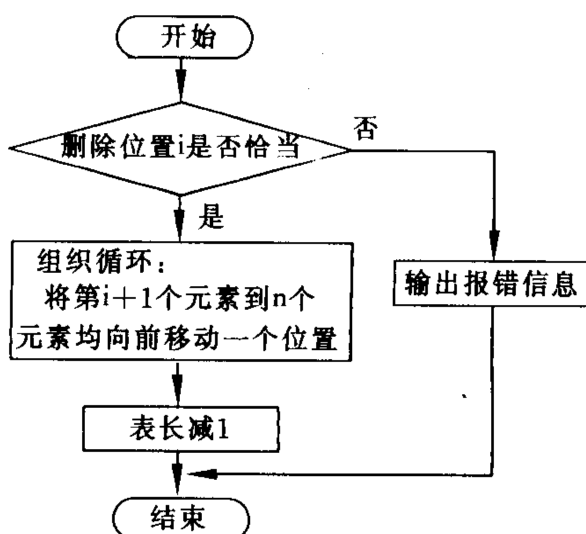


图 3-5　线性表顺序存储结构的删除算法框图

3.3　线性表的链式存储结构

上一节我们讨论了线性表的顺序存储结构，其特点是逻辑关系上相邻的两个元素在物理位置上也相邻。本节我们将讨论另一种存储结构——链式存储结构，它不要求逻辑上相邻的元素在物理位置上也相邻。

一、线性链表

线性表的链式存储结构是用一组任意的存储单元存储线性表中的数据元素；这组存储单元可以是连续的，也可以是不连续的。因此，为了表示数据元素 a_i 与其直接后继元素 a_{i+1} 之间的逻辑关系，对于数据元素 a_i 来说，除了存储元素本身的信息之外，还需存储指示其直接后继的信息(即直接后继的存储位置)。这两部分信息组成数据元素 a_i 的存储映象，称为结点(node)，它包含两个域：一个域用于存储数据元素的信息称为数据域；另一个域用于存储直接后继元素的存储位置称为指针域(或链域)。在 PASCAL 语言中可以用如下的指

针类型描述一个结点：

```
TYPE pointer = ↑nodetype;
     nodetype = RECORD
                  data : elemtp;
                  next : pointer;
                END;
```

其中，pointer 是一个指针类型，nodetype 表示结点类型。在一个结点中有两个域：data 域用于表示数据元素，其类型为 elemtp，elemtp 根据实际而定，可以是一个整型数，也可以是一个字符等；next 域是指针域，用于指示直接后继元素的存储位置。这样由 n 个数据元素的 n 个结点通过 next 域链结成一个链表，即为线性表$(a_1, a_2, \cdots, a_n)$的链式存储结构即链表。由于此链表的每个结点只含一个指针域，故又称为线性链表或单链表。

例如：有一线性表(49,38,65,98,27,16)，它的线性链表存储结构如图 3-6 所示。

Head: 38

存储地址	数据域	指针域
1	27	16
9	65	74
16	16	NIL
25	38	9
38	49	25
74	98	1

图 3-6 线性链表的物理状态图示

由此我们可以看出，在线性表的链式存储结构中，数据元素之间的逻辑关系是由线性链表中结点的指针域表示。换句话说，指针是数据元素之间逻辑关系的映象，而结点在存储器中的位置是可以任意按排的。另外，在线性链表中必须指出第一个元素的存储地址，所以需要设立一个特殊的指针，称为头指针(head)。而由于最后一个元素没有直接后继，所以它的指针域的值为“空”(nil)。在线性链表中，当头指针为“空”(head=nil)时，表示线性表为空表。

由于如图 3-6 表示的线性链表，数据元素的逻辑顺序不易观察；而在实际应用中我们往往只关心线性表中元素的逻辑顺序，而不是元素在存储器中的位置。所以，通常我们把线性链表用图 3-7 所示的形式表示。

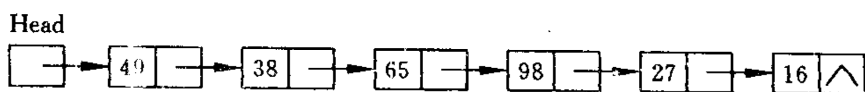


图 3-7 线性链表的逻辑状态图示

其中，结点之间的箭头表示指针域，符号"∧"表示 nil。有时，我们在线性链表的第一个结点之前附设一个结点，称之为头结点。头结点的数据域可以不存储任何信息，也可用于存储诸如线性表长度等附加信息；头结点的指针域用于存储第一个元素的存储位置。如图 3-8(a)所示，此时线性链表的头指针指向头结点。若线性表为空表，则头结点的指针域的值为"空"，如图 3-8(b)所示：

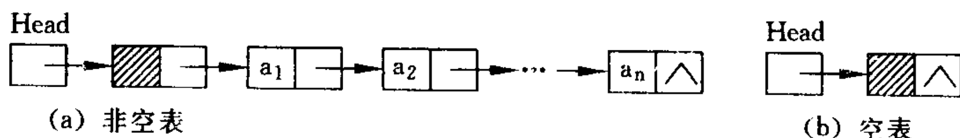


图 3-8 带头结点的线性链表图示

对任何一种数据结构来说，不管采用什么样的存储结构，都必须能表示出元素之间的逻辑关系；对存储结构的选择，往往是以提高特定操作的效率为依据。

二、线性链表的插入

若有线性表(a_1,a_2,…, a_n)，用带头结点的线性链表存储，表头指针为 head，要求在第 i 个元素之前插入一个新的元素 b。设 p 为指向 a_i 的直接前驱结点 a_{i-1}的指针。插入时，首先要生成一个新的结点由 s 指向，在 s 所指结点的数据域中存入 b，再令 s 结点的的指针域指向存储元素 a_i 的结点(由 p↑.next 指向)；然后使存储元素 a_{i-1}的结点的指针域(p↑.next)指向 s。线性链表在执行插入操作前后的逻辑状态如图 3-9 所示：

这种插入操作只改变了两个结点的指针域，并未对数据元素作任何移动。当然在插入之前首先要搜索到元素 a_i 的直接前驱结点，并判别插入位置是否合理。图 3-10 为插入算法的框图描述。

如下是线性链表插入算法的 PASCAL 语言描述。

```
PROCEDURE ins-linklist(head:pointer;i:integer;b:elemtp);
```

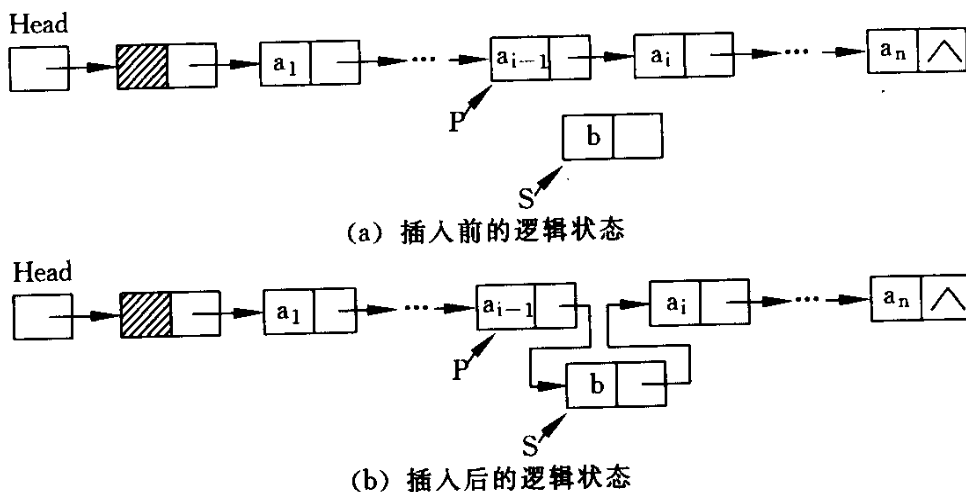


图 3-9 线性链表插入操作的逻辑状态图示

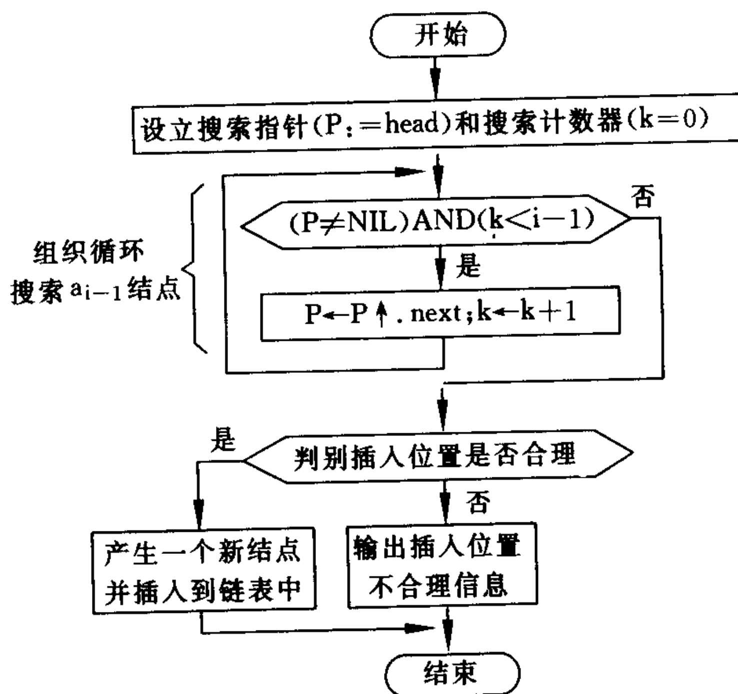


图 3-10 线性链表插入算法的框图

```
VAR p,s : pointer;
      k : integer;
BEGIN
    p:=head; k:=0;
    WHILE (p<>nil) AND (k<i-1) DO
      BEGIN p:=p↑.next; k:=k+1 END;
    IF (p=nil) THEN writeln ('No this postion!')
```

```
ELSE BEGIN
          new(s); s↑.data:=b;
          s↑.next:=p↑.next;
          p↑.next:=s
        end
END
```

三、线性链表的删除

在一个线性表($a_1,a_2,\cdots,a_n$)中删除第 i 个元素。若线性表用带头结点的线性链表存储，则删除第 i 个元素(a_i)时，必须改变第 i−1 个元素(a_i 的直接前驱元素 a_{i-1})结点的指针域，使其指向第 i+1 个元素(a_i 的直接后继元素 a_{i+1})的结点，然后把第 i 个元素的结点归还给系统。设 p 为指向元素 a_{i-1}结点的指针，q 为指向元素 a_i 结点的指针，则线性链表执行删除操作前后的逻辑状态如图 3-11 所示：

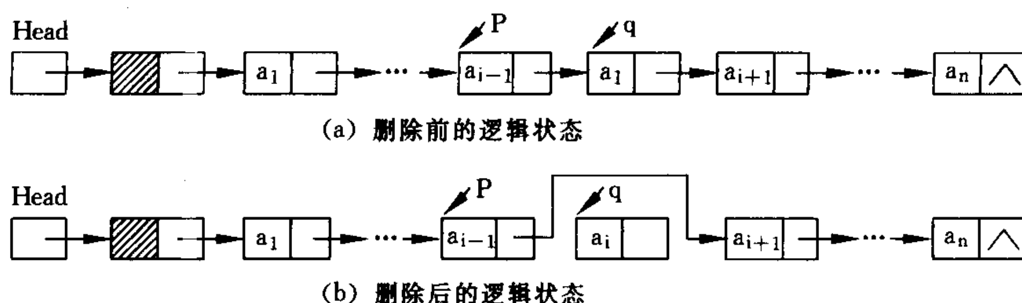


图 3-11 线性链表删除操作的逻辑状态图示

同插入操作一样，在这里也不需要移动数据元素，只需要搜索到第 i−1 个元素结点，并判别删除位置是否合理，然后再进行删除，其算法的框图描述如图 3-12 所示。

四、循环链表

循环链表是线性表的另一种形式的链式存储结构，它与线性链表的不同之处在于：在线性链表中，最后一个结点的指针域为“空”(nil)；而在循环链表中最后一个结点的指针域指向头结点，使整个链表形成一个环。所以，从表中任一结点出发都可以找到其它结点。

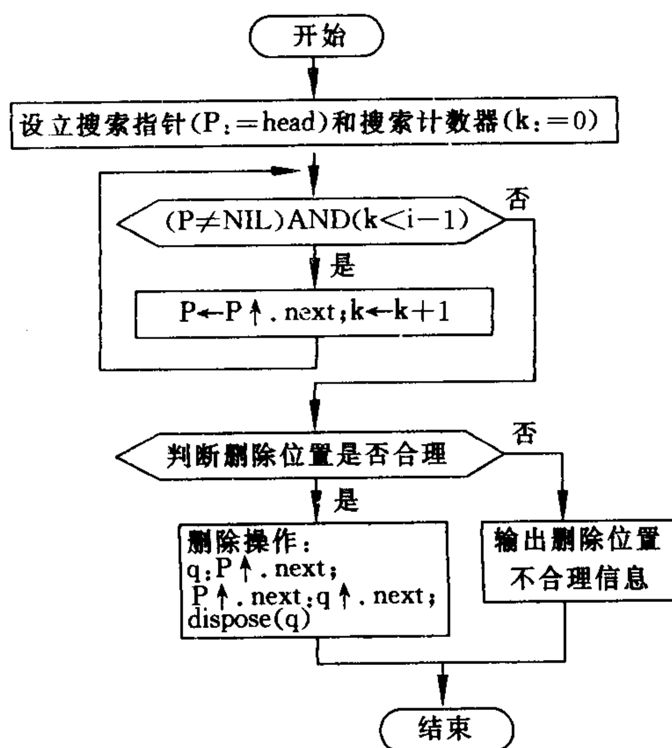


图 3-12 线性链表删除算法的框图

如图 3-13 所示为循环链表的逻辑状态。

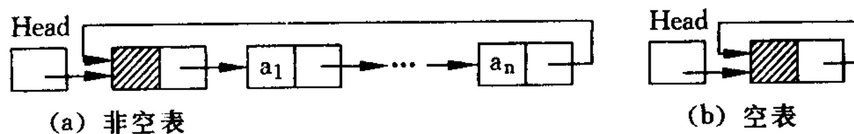


图 3-13 带头结点的单循环链表

在循环链表上的操作和线性链表基本一样，差别仅在于表尾结点的判别条件：在线性链表中，是根据结点的指针域是否为“空”(nil)进行判别；而在循环链表中，是根据结点的指针域是否指向头结点进行判别。

五、双向链表和循环双向链表

在线性链表和循环链表的结点中，只有一个指向直接后继结点的指针域，所以从一个结点出发只能根据指针往后搜寻其他结点，而不能直接搜寻结点的直接前驱结点。若要寻找一个结点的直接前驱，则需从头指针开始搜寻。为了克服线性链表单向性的缺点，人们提出

了双向链表的概念。

所谓双向链表就是在链表的每个结点中除了设置数据域以外，再有两个指针域，其一和单链表一样用于指向直接后继结点，另一个则用于指向直接前驱结点。结点的结构如图3-14所示：

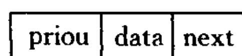


图 3-14　双向链表中的结点

可用 PASCAL 语言描述如下：

```
TYPE dupointer = ↑dunodetype;
      donodetype=record
                    data:elemtp;
                    priou,next:dupointer
                   end;
```

其中，priou 为指向前驱结点的指针，称为前驱指针；next 为指向后继结点的指针，称为后继指针，data 域同单链表一样。双向链表的逻辑状态如图 3-15 所示。其中，表尾结点的后继指针为“空”，头结点的前驱指针为“空”。

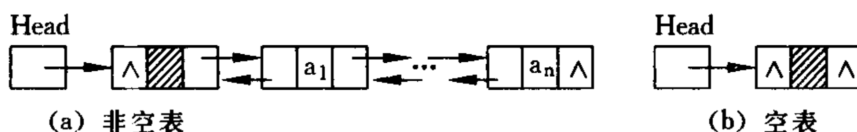


图 3-15　双向链表的逻辑状态

在双向链表中，插入操作和删除操作都需要同时修改两个方向上的指针。图 3-16 为插入操作完成前后有关结点的指针修改情况。其中，新插入的结点由 s 指向，结点插在元素 a_i 结点的前面，a_i 结点由 p 指向。

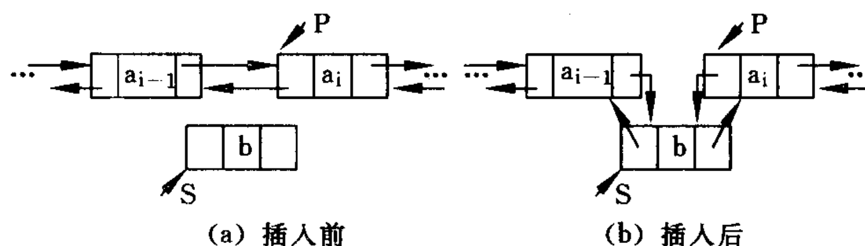


图 3-16　双向链表插入操作的逻辑状态

图 3-17 是删除操作完成前后有关结点的指针修改情况。其中被删除元素 a_i 的结点由 p 指向。

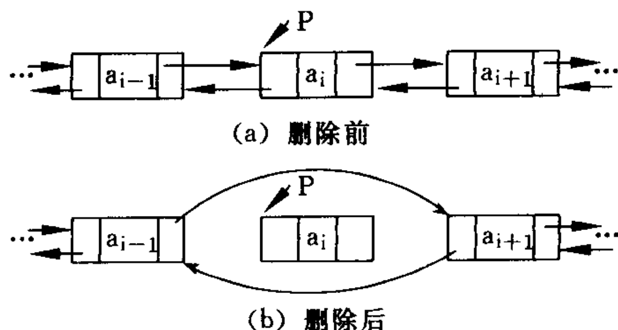


图 3-17　双向链表删除操作的逻辑状态

在双向链表中执行插入操作和删除操作的有关算法由读者自己思考,要注意指针域修改的先后顺序。

同单链表类似,双向链表也可以构成循环链表,并称为循环双向链表。其特点就是表尾结点的后继指针指向头结点;头结点的前驱指针指向表尾结点。如图 3-18 所示,为循环双向链表的逻辑状态。

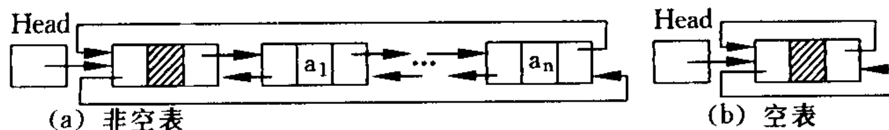


图 3-18　循环双向链表的逻辑状态

在循环双向链表上的插入操作和删除操作与双向链表上的相应操作类似。

3.4　一元多项式的相减

在这一小节中,我们以一元多项式相减问题为例,讨论有关线性表的应用。

一、一元多项式的表示

在数学上,一元 n 次多项式 $P_n(x)$ 可按升幂序写成:

$$P_n(x)=p_0+p_1*x+\cdots+p_n*x^n$$

它由 n+1 个系数唯一确定。所以,在计算机中,可以用一个线性表 P

来表示：

$$P=(p_0,p_1,\cdots,p_n)$$

其中，每一项的指数 i 隐含在其系数 p_i 的序号中（在这里设第一元素的序号为 0）。显然，我们可以对 P 采用顺序存储结构，这样就使得多项式相减算法的定义十分简洁。例如，假设 $Q_m(x)$是一元 m 次多项式，同样可以用线性表 Q 表示：

$$Q=(q_0,q_1,\cdots,q_n)$$

不失一般性，设 $m<n$，则多项式 $P_n(x)$减 $Q_m(x)$的结果 $R_n(x)=P_n(x)-Q_m(x)$可用线性表 R 表示：

$$R=(p_0-q_0,p_1-q_1,\cdots,p_m-q_m,p_{m+1},\cdots,p_n)$$

至此我们可以看出，一元 n 次多项式的这种表示很明确，操作也很方便。但是，在实际应用中，多项式的幂次数可能很高，并且许多系数可能为零；所以使用这种表示方法，在线性表中必然存在许多零元素，将它们存放在内存中必然浪费空间。例如：有一多项式 S(x)：

$$S(x)=1+3x^{1000}-2x^{2000}$$

则它的线性表长度为 2001。其中，零元素有 1998 个，而非零元素只有 3 个。对这个线性表，不管采用顺序存储结构还是链式存储结构，都将造成内存空间的极大浪费。为此，我们设想用另外一种二元组（一个元素有两个数据项）的形式表示多项式，使之只保存非零系数项。当然，此时必须同时保存非零系数以及相应的指数。

一般情况下，一元 n 次多项式 $P_n(x)$可写成：

$$P_n(x)=p_1x^{e_1}+p_2x^{e_2}+\cdots+p_mx^{e_m}$$

其中，　　$pi<>0(i=1\cdots m),0\leqslant e_1<e_2\cdots<e_m=n$。可以用一个线性表表示这样的多项式：

$$((p_1,e_1),(p_2,e_2),\cdots,(p_m,e_m))$$

其中，每个元素有两个数据项（系数，指数）。

这种表示方法，在最坏情况下 n+1 个系数都不为 0，此时上面的方案要比前一方案多存储一倍的数据。但是，对于 S(x)这类多项式，则这种表示将大大节省内存。

当然，对于这个线性表，可以用顺序存储结构存储（如图 3-19 所

示），也可以用链式存储结构存储。如图3-20所示。

p_1	e_1
p_2	e_2
⋮	⋮
p_m	e_m

图 3-19　多项式二元组线性表的顺序存储结构

它们的 PASCAL 语言描述分别为：

```
TYPE elemtp = RECORD
                coef:real;
                exp :integer;
              END;
polynty=ARRAY[1..m] OF elemtp;
```

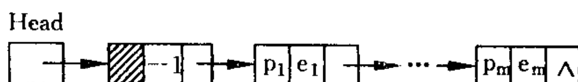


图 3-20　多项式二元组线性表的单链表存储结构

这是顺序存储结构的描述。其中，二元组(coef,exp)分别表示系数和指数。

```
TYPE polylink = ↑nodetp;
     nodetp = RECORD
                coef:real;
                exp:integer;
                next:polylink;
              END;
```

这是单链表存储结构的结点描述。

在实际应用中，究竟采用哪一种存储结构，则要根据具体的应用而定。例如，求多项式的值，在运算过程中只需访问多项式的系数和指数，则选择顺序存储结构为宜；又如求两个多项式的和、差、积等运算，则由于在执行加法、减法、乘法等操作时，非零系数会经常修改，线性表要经常进行插入和删除操作，所以应采用链式存储结构为宜。下面我们将在带头结点的线性链表上讨论两个一元多项式的减法运算。

二、多项式相减运算

两个一元多项式相减，运算规则很简单：对于两个多项式中指数相同的项，其相应的系数相减，若差不为零，则构成“差多项式”中的

一项；对于在被减多项式中存在而在减数多项式中不存在的项，直接复抄到“差多项式”中；对于在减数多项式中存在而在被减多项式中不存在的项，则需要改变系数符号后，加到“差多项式”中。

在这里以带头结点的线性链表作为存储结构，并且结果多项式保留在被减多项式上，且不改变减数多项式的线性链表。例如，有两个一元多项式：

$$A(x)=5+3x+9x^6+5x^{14}$$

$$B(x)=4x+9x^6-8x^8$$

要求计算 $A'(x)=A(x)-B(x)$。图 3-21 是 A(x)、B(x)的线性链表存储结构。

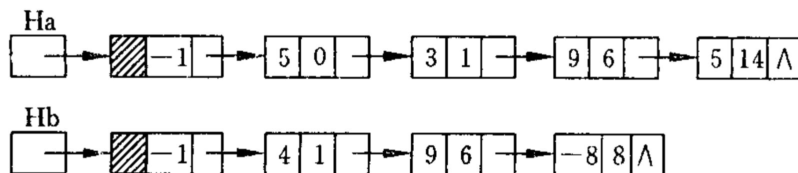


图 3-21　多项式 A(x)、B(x)的线性链表

假设两个线性链表的头指针分别为 ha、hb。为了实现两个多项式相减运算，需要设立两个搜索指针 pa 和 pb 分别指向当前被检测的结点(开始时 pa:=ha↑.next;pb:=hb↑.next)。则相减的步骤可概括为：依次取 A(x)、B(x)中的结点进行比较，并根据比较结果执行下面操作：

1. 若 pa=nil 或 pa↑.exp>pb↑.exp，则需在 A 表的表尾或 pa 结点的前面插入一个系数为－pb↑.coef、指数为 pb↑.exp 的结点。然后 pb 向后推进一步(pb:=pb↑.next)，pa 不变。

2. 若 pa<>nil 并且 pa↑.exp=pb↑.exp，则

当 pa↑.coef－pb↑.coef<>0 时，需要修改 pa↑.coef(pa↑.coef:=pa↑.coef－pb↑.coef)，且 pa、pb 同时向后推进一步(pa:=pa↑.next;pb:=pb↑.next)。

当 pa↑.coef－pb↑.coef=0 时，需把 pa 结点删除，且 pa、pb 同时向后推进一步。

3. 若 pa<>nil 并且 pa↑.exp<pb↑.exp 则只需 pa 向后推进

一步。如此直到B(x)的非零系数都被运算过(即 pb=nil)。

由于在运算中需要在 pa 前插入结点或删除 pa 结点,所以需修改 pa 的前驱结点的指针域,为此还需设立一个辅助指针 qa 以指示 pa 的前驱结点。图 3-22 是多项式相减算法的框图描述。

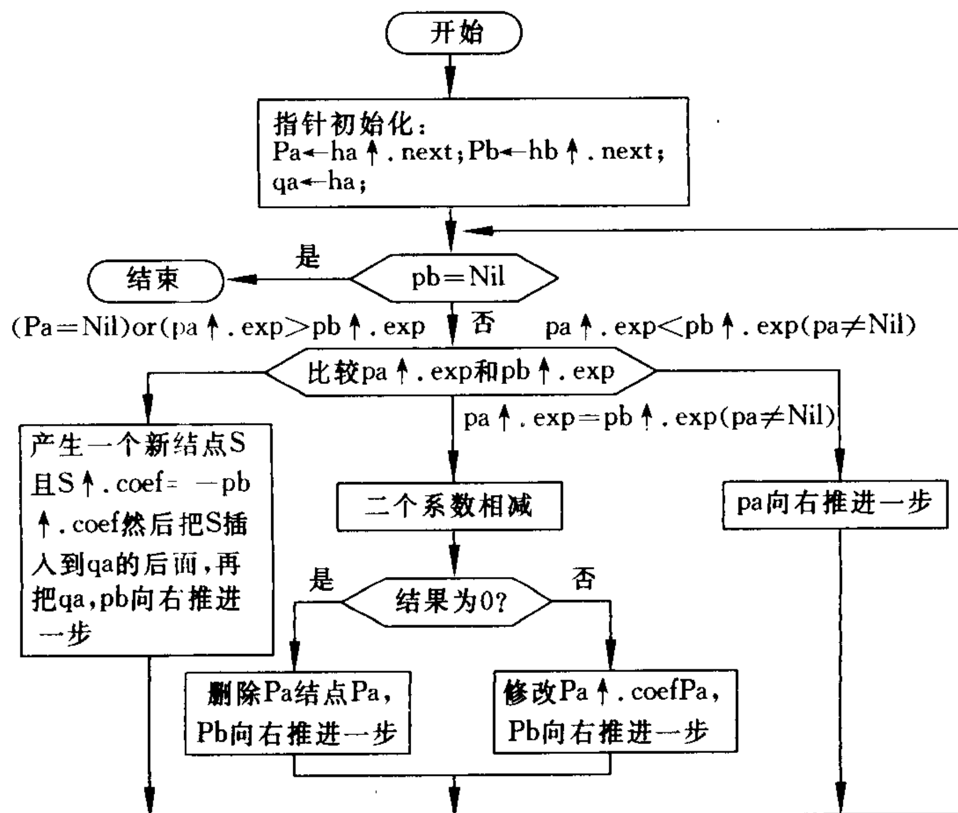


图 3-22　多项式相减算法框图

该算法的 PASCAL 语言描述如下:

```
PROCEDURE ploysub(VAR ha:ploylink;hb:ploylink);
  VAR pa,pb,qa,s:ploylink;
  BEGIN
      pa:=ha↑.next; pb:=hb↑.next; qa:=ha;
      WHILE pb<>nil DO
        IF (pa=nil) OR (pa↑.exp>pb↑.exp)
          THEN BEGIN
                  new(s);s↑.coef:=-pb↑.coef; s↑.exp:=pb↑.exp;
                  s↑.next:=pa; qa↑.next:=s;
                  qa:=s; pb:=pb↑.next
```

```
            END
        ELSE
        IF pa↑.exp=pb↑.exp
          THEN BEGIN
                 pa↑.coef:=pa↑.coef-pb↑.coef;
                 IF pa↑.coef=0
                   THEN BEGIN
                          qa↑.next:=pa↑.next;
                          dispose(pa);
                          pa:=qa↑.next;
                          pb:=pb↑.next
                        END
                   ELSE BEGIN
                          qa:=pa; pa:=pa↑.next;
                          pb:=pb↑.next
                        END
                 END
            ELSE BEGIN
                   qa:=pa; pa:=pa↑.next
                 END
END;
```

假设多项式 A(x)中有 m 项,B(x)中有 n 项,则这个算法的时间复杂度为 O(m+n)。

习　题

1. 试分别以顺序存储结构和链式存储结构实现线性表的就地倒置算法，即在原表的存储空间内将线性表$(a_1,a_2,\cdots,a_n)$倒置为$(a_n,a_{n-1},\cdots,a_1)$。
2. 设有两个线性表 $X=(x_1,x_2,\cdots,x_n)$, $Y=(y_1,y_2,\cdots,y_m)$。试写一合并 X,Y 为线性表 Z 的算法，使得：

$$Z=\begin{cases}(x_1,y_1,\cdots,x_m,y_m,x_{m+1},\cdots,x_n), & 当\ m<=n;\\(x_1,y_1,\cdots,x_n,y_n,y_{n+1},\cdots,y_m), & 当\ m>n;\end{cases}$$

要求X、Y,Z用链表存储，并且Z表利用X和Y中的结点空间。

3. 写一个算法，求出线性表中数据域的值为x的结点序号。序号从表头算起,若链表中没有此结点则序号为零。

4. 试以循环链表作为稀疏多项式的存储结构，编写求其导函数的算法，要求利用原多项式的结点空间存放结果多项式并释放无用结点。

第四日　栈和队列

栈和队列是两种重要的线性结构，它们在各种软件系统中被广泛的应用。从数据结构角度看，栈和队列也是线性表，数据元素之间存在线性关系，只是栈和队列的基本操作是线性表的子集，它们是操作受限的线性表。本日将主要讨论栈和队列的定义、表示方法及应用。

4.1　栈

一、栈的定义

所谓栈就是限定仅在表的同一端进行插入数据元素（入栈操作）或删除数据元素（出栈操作）的线性表。允许插入数据元素和删除数据元素的一端称为栈顶（top），而表中固定的一端称为栈底（bottom）。不含任何元素的栈称为空栈。

假设有个栈 $S=(a_1,a_2,\cdots,a_n)$，则 a_1 为栈底元素，a_n 为栈顶元素。由于栈只允许在栈顶进行插入和删除元素，所以它们进栈的次序为 $a_1,a_2,\cdots,a_n$，而出栈的次序为 $a_n,a_{n-1},\cdots,a_1$。可见栈的操作是按后进先出的原则进行的，如图 4-1 所示。因此，栈又称为后进先出（Last in First out）的线性表（简称 LIFO 表）。在日常生活中有许多类似于栈的例子。例如在桌子上有一叠碗，规定每次只能放一只碗在上面，并且放在这叠碗的顶上；每次只能取一只

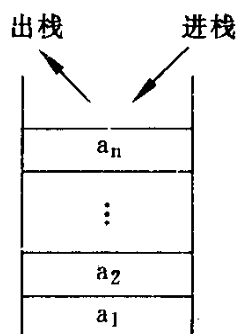


图 4-1　栈的示意图

碗，并且从这叠碗的顶上拿。这就很类似于栈，放碗时相当于入栈操作，取碗相当于出栈操作。

二、栈的基本操作

栈的基本操作除了插入（入栈）操作和删除（出栈）操作外，还有栈的初始化、判定栈空和取栈顶元素等。

1. 初始化(inistack(S))：设定S为一个空栈。
2. 判栈空(empty(S))：若栈S为空，则返回值TRUE；否则返回值FALSE。
3. 入栈(push(S,x))：把元素x插入到栈s的栈顶。
4. 出栈(pop(S))：若栈S不空时，把栈顶元素弹出，并返回其值；否则返回一个"空"值。
5. 取栈顶元素(gettop(S))：当栈S不空时返回栈顶元素值；否则返回"空"值。注意它和出栈操作的区别在于它不弹出栈顶元素。

三、栈的存储结构

栈的存储结构同线性表一样有两种：顺序存储和链式存储，下面将分别介绍。

1. 栈的顺序存储结构

栈的顺序存储结构就是用一组连续的存储单元依次存放自栈底到栈顶的数据元素，同时设立指针top（称为栈顶指针）以指示栈顶元素的当前位置。假设用一维数组S[1..arrmax]表示栈，则当top=arrmax时表示栈满，此时若有元素入栈则将产生"数组越界"的错误称之为"上溢"；反之，当top=0时表示栈空，此时若要进行元素出栈操作则也将产生"数组越界"的错误称之为"下溢"。图4-2表示了栈顶指针同栈中元素之间的关系。

在PASCAL语言中，可以用如下定义描述栈的顺序存储结构：

```
CONST arrmax={栈中允许存放元素的最大数}；
TYPE sqstktp=RECORD
```

```
    elements : ARRAY [1..arrmax] OF elementtype;
    top : 0..arrmax
  END;
```

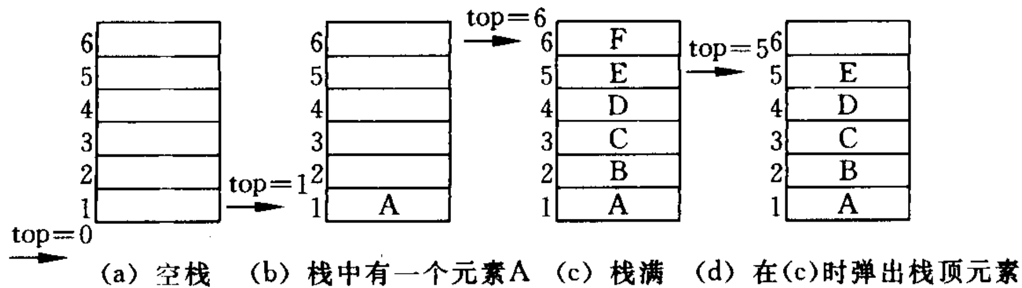


图 4-2　栈顶指针与栈中元素的关系

其中,数据域 elements 描述了栈的存储空间,数据域 top 为栈顶指针。在这种存储结构上,栈的五种基本操作都很容易实现。下面直接给出它们的 PASCAL 语言描述。

```
PROCEDURE inistack(VAR s:sqstktp);
BEGIN
  s.top:=0
END;

FUNCTION push(VAR s:sqstktp; x:elementype) : boolean;
BEGIN
  IF s.top=arrmax
    THEN return(FALSE)
    ELSE BEGIN
           s.top:=s.top+1;
           s.elements[s.top]:=x;
           return(TRUE)
          END
END;

FUCTION pop(VAR s:sqstktp) : elementype;
BEGIN
  IF s.top=0
```

```
      THEN return(NULL)
      ELSE BEGIN
             s.top:=s.top-1;
             return(s.elements[s.top+1])
            END
END;

FUCTION empty(s:sqstktp):boolean;
BEGIN
  IF s.top=0
    THEN return(TRUE)
    ELSE return(FALSE)
END;

FUNCTION gettop(s:sqstktp):elementtype;
BEGIN
  IF s.top=0
    THEN return(NULL)
    ELSE return(s.elements[s.top])
END;
```

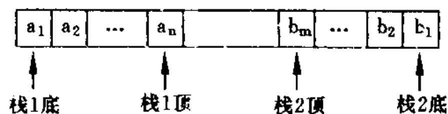


图 4-3　两个栈共享空间示意图

在实际工作中,栈的应用非常广泛,特别是在一些大型软件系统中,往往会同时使用多个栈。当然,这时可以为每一个栈安排一个数组,但这样做并不实际。因为各个栈的实际使用空间在使用期间是不断变化的,常常会有这样的情况:其中某一栈"上溢"时,而另外的栈还有许多的空闲空间。所以,如果能使多个栈共享空间,则将提高空间的使用效率,从而减少发生栈的"上溢"。

例如在实际应用中需要设立两个栈时,可以使它们共享一维数组空间 s[1..arrmax],两个栈的栈底分别设在数组的两端。入栈操作时都向中间延伸,仅当两个栈的栈顶指针在中间相遇时才发生"上溢",如图 4-3 所示。

从中可以看出，由于两个栈之间的存储空间可以互补，使得每个栈实际可利用的最大空间大于 arrmax/2。当在实际应用中需要设立多个栈时，也可以使它们共享一个数组，但操作要比刚才复杂一些，此时最好采用下面介绍的链式存储结构。

2. 栈的链式存储结构

对于一个栈，我们也可以用一个线性链表作为存储结构。用链表表示的栈简称为链栈，其逻辑结构如图 4-4 所示。

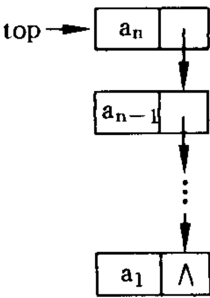


图 4-4　链栈示意图

其中，top 为栈顶指针。可用 PASCAL 语言作类型说明如下：

```
TYPE linkstack=↑stknode;
     stknode=RECORD
               data:elemtype;
               next:linkstack
             END;
```

对于链栈，当 top=nil 时表示栈空；而栈满的情形只有当计算机系统中的可利用空间都被占用时才会发生。所以多个链栈要么同时发生栈满情形；而不会发生其中之一栈满，而其它栈有空闲空间的情形。显然，多个链栈共享空间也就是自然而然的事了。对于链栈来说，栈的基本操作是很容易实现的，读者可自行完成之。

4.2　队　　列

一、队列的定义

队列是限定在表的一端进行插入数据元素（入队操作）、在表的另一端进行删除数据元素（出队操作）的线性表。其中，允许插入的一端称之为队尾(rear)，允许删除的一端称之为队头(front)。当队列中不含任何数据元素时称为空队列。

假设有一个队列 $Q=(a_1,a_2,\cdots,a_n)$，则 a_1 称为队头元素，a_n 称为

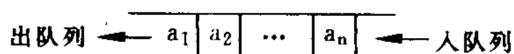


图 4-5 队列示意图

队尾元素。元素入队的次序为 $a_1,a_2,\cdots,a_n$，出队的次序也为 $a_1,a_2,\cdots,a_n$。可见队列的操作是按先进先出原则进行的，如图 4-5 所示。因此，队列又称为先进先出(First In First Out) 的线性表(简称为 FIFO 表)。这就如同日常生活中排队买东西一样，最早进入队列的最先买到东西离开队列。

二、队列的基本操作

队列的基本操作同栈类似，也有五种，不同的是删除数据元素是在队头进行。

1. 初始化(iniqueue(Q))：设定 Q 为一个空队列。
2. 判队列空(empty(Q))：判别队列 Q 是否为空。若队列为空则返回值 TRUE；否则，返回值 FALSE。
3. 入队列(enqueue(Q,x))：把数据元素 x 插入到队列 Q 的队尾。
4. 出队列(delqueue(Q))：若队列 Q 不空，则把队头元素删除，并返回其值；否则，返回一个"空"值。
5. 取队头元素(gethead(Q))：若队列 Q 不空，则返回队头元素；否则，返回一个"空"值。

三、队列的存储结构

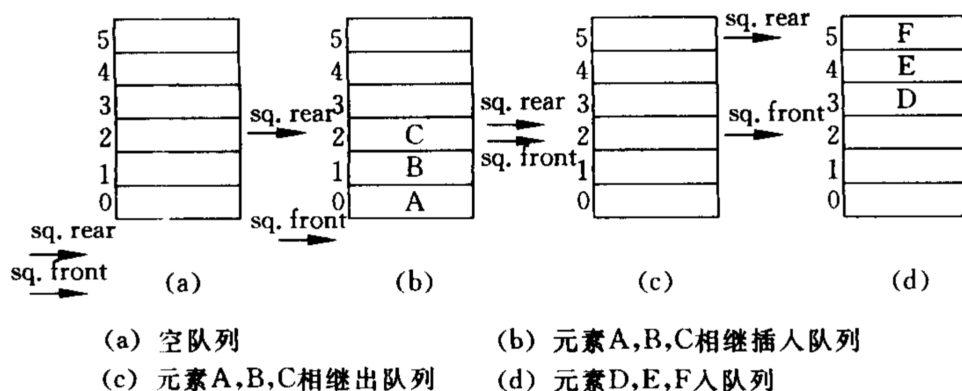
1. 队列的顺序存储结构——循环队列

同栈的顺序存储结构类似，在队列的顺序存储结构中，需要用一组连续的存储单元依次存放自队头到队尾的数据元素，并且还需设立两个指针分别指示队头元素和队尾元素。在此约定：队尾指针指示队尾元素在队列中的当前位置，队头指针指示队头元素的前一个位置。在 PASCAL 语言中可用如下的类型定义描述队列的顺序存储结构：

```
CONST maxsize={队列的最大容量}；
```

```
TYPE cyclicquetp=RECORD
                    elements:array[0..maxsize-1] OF elementtype;
                    rear,front:0..maxsize-1;
                 END;
```

其中,rear 和 front 分别表示队尾和队头指针,elements 为存储队列元素的向量。图 4-6 展示了在队列的顺序存储结构中数据元素和队头、队尾指针的关系。



(a) 空队列　(b) 元素A,B,C相继插入队列
(c) 元素A,B,C相继出队列　(d) 元素D,E,F入队列

图 4-6　队列顺序存储结构中队头、队尾指针与元素的关系

从中可以看出,队列为空时,队头与队尾指针之间存在如下关系(设 sq 为 cyclicquetp 变量):

sq.front=sq.rear

如图 4-6 中(a)和(c)所示的情况。在队列的顺序存储结构中,需要讨论的是队列满(上溢)的判定条件是什么?在图 4-6(d)中队尾指针已指向向量的上界。若要进行入队操作(sq.rear:=sq.rear+1; sq.elements[sq.rear]:=x) 必然发生"上溢"。但实际上向量中还有三个空的存储单元。这种现象称之为"假溢出"。对于这一问题,可以用下面两种方法解决:其一是当发生"假溢出"时,将全部元素向前移动,使队头元素存放在向量下界的位置,当然,移动元素需要花费一定的时间;其二是把存储队列元素的向量 elements[0..maxsize-1]看成一个循环表(elements[0]接在 elements[maxsize-1]的后面)。这种队列称为循环队列,使用比较方便。图 4-7 为循环队列的示意图。

在循环队列中,入队操作可以如下描述(不考虑上溢):

sq.rear:=(sq.rear+1) MOD maxsize;

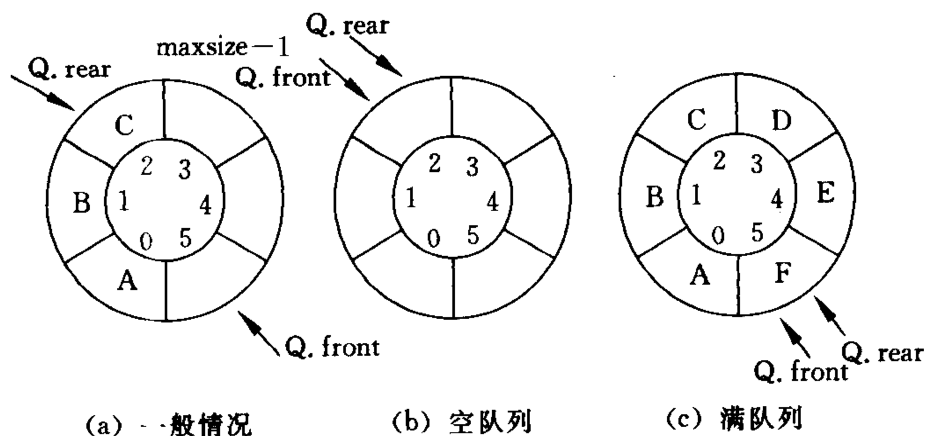


图 4-7　循环队列示意图

sq. elements[sq. rear]:＝x;

出队操作可以简单地如下描述:

sq. front:＝(sq. front＋1) MOD maxsize;

在循环队列中,虽然很巧妙地解决了“假上溢”的问题。然而,随之又产生了一个新的问题——如何判别队列空和队列满?

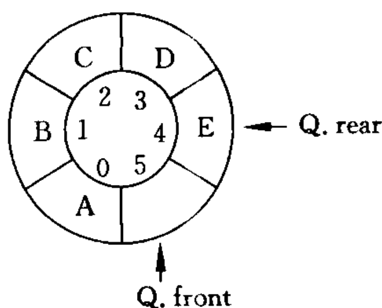


图 4-8　队满示意图

假设当前状态为图 4-7(a)所示,现有三个元素 D、E、F 入队列,使队列呈满的状态。此时,q. front＝q. rear,如图 4-7(c)所示。假设在图 4-7(a)所示状态时,A、B、C 三个元素相继出队列,则队列呈空状态,如图 4-7(b)所示,此时也有 q. front＝q. rear。由此可见,只凭 q. front＝q. rear 还不能判定循环队列是空还是满。对于这个问题我们可以用这样的方法解决:队列仍然设头指针和尾指针,并且当 q. front＝q. rear 时表示队列空;只是将队列满的判别条件约定为(q. rear＋1) MOD maxsize＝q. front。也就是说,当执行入队操作时,队尾指针从后面赶上队头指针就认为队满,发生“上溢”。图 4-8 就是队满示意图。

在这里可以看到头指针所指示的存储单元永远是空闲的。当然,对于队满和队空的判别也可以通过另外设立一个标志进行,但这样做操作时就要多花一些时间。

采用上述存储结构，队列的基本操作都很容易实现。下面仅就出队和入队操作给出框图描述和 PASCAL 语言描述。见图 4-9 及图 4-10。

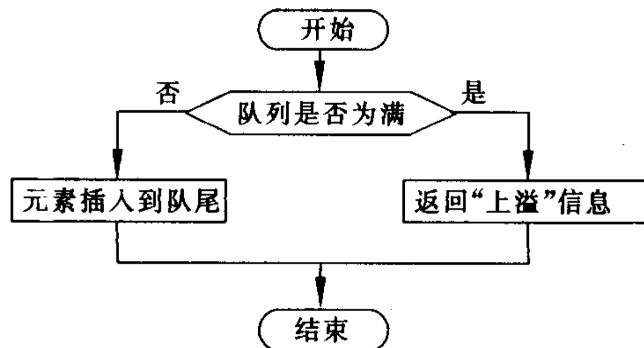


图 4-9　循环队列入队操作的算法框图

```
FUNCTION en-cyque(VAR q:cyclicquetp; x:elementype):boolean;
BEGIN
  IF (q.rear+1) MOD maxsize=q.front
    THEN return(FALSE)
    ELSE BEGIN
           q.rear:=(q.rear+1) MOD maxsize;
           q.elements[q.rear]:=x
         END
END;
```

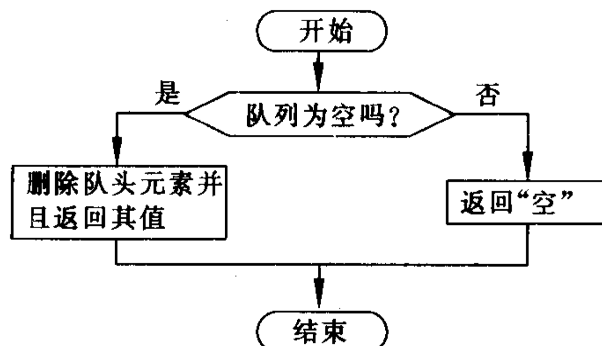


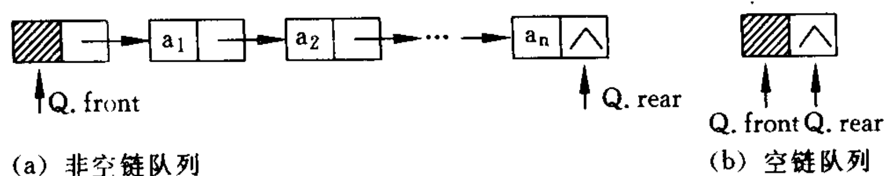
图 4-10　循环队列出队操作的算法框图

```
FUNCTION dl-cyque(VAR q:cyclicquetp):elementype;
  BEGIN
    IF q.rear=q.front
```

```
        THEN return(NULL)
        ELSE BEGIN
                q.front:=(q.front+1) MOD maxsize;
                return(q.elements[q.front])
              END
    END;
```

2. 队列的链式存储结构

队列和栈一样，需要经常进行插入和删除操作，所以数据元素的变动较大。为此，用链式存储结构比顺序存储结构更为合适。用链表表示的队列简称为链队列。



(a) 非空链队列　　(b) 空链队列

图 4-11　链队列示意图

在链队列中，用一个带头结点的单链表存储队列元素，并设置两个指针(头指针和尾指针)分别指向头结点和队尾结点；当队列为空时，头指针和尾指针均指向头结点。其逻辑结构如图 4-11 所示，其 PASCAL 语言的类型定义如下：

```
    TYPE queueptr= ↑queuenode;
        queuenode=RECORD
                    data:elementtype;
                    next:queueptr
                  END;
        linkquetp=RECORD
                    front,rear:queueptr
                  END;
```

在链队列上，队列的基本操作也很容易实现。下面对初始化、入队、出队操作给出其算法的 PASCAL 语言描述。

```
    PROCEDURE init-linkque(VAR q:linkquetp);
    BEGIN
      new(q.front); q.rear:=q.front; q.front↑.next:=nil;
```

```
END;

FUNCTION en—linkque(VAR q:linkquetp; x:elementype):boolean;
BEGIN
  new(q.rear↑.next);
  q.rear:=q.rear↑.next;
  q.rear↑.data:=x;
  q.rear↑.next:=nil;
END;

FUNCTION dl—linkque(VAR q:linkquetp):elementype;
VAR s:queueptr;
      x:elementype;
BEGIN
  IF q.front=q.rear
    THEN return(NULL)
    ELSE BEGIN
           s:=q.front↑.next;
           q.front↑.next:=s↑.next;
           IF s↑.next=nil THEN q.rear:=q.front;
           x:=s↑.data; dispose(s);
           return(x)
         END
END;
```

链队列同链栈的情况相同，一般情况下不会发生"上溢"现象。除非是整个系统的可利用空间都被占满。

4.3 算术表达式的计算

算术表达式的计算是程序设计语言编译中的一个最基本的问题；它的实现是栈应用的典型例子。在这里介绍一种简单直观、广为使用的算术表达式的计算算法——算符优先算法。

要把一个算术表达式翻译成能正确求值的一组机器指令序列，或者直接对表达式进行计算求值，首先要能够正确解释表达式。例如，对下述算术表达式求值：

$$4+5*(6*2)/4$$

首先要知道算术运算的规则：(1).先乘除、后加减；(2).先括号内、后括号外；(3).从左算到右。因此这个算术表达式的计算顺序为：

$$4+5*(6+2)/4=4+5*8/4=4+40/4=4+10=14$$

算符优先算法就是根据这个运算规则来实现对算术表达式的编译或计算。

任何一个算术表达式都是由操作数，运算符和界符组成。在这里为了叙述的简洁，我们仅讨论一种简单算术表达式的求值。这种表达式中只含加、减、乘、除四种运算符，界符为左右括号，操作数为整数。我们把运算符和界符统称为算符。根据上述三条运算规则，在运算的每一步中，任意两个相继出现的算符 OP_1 和 OP_2 之间的优先关系至多为下面三种关系之一；

$OP_1<OP_2$　　表示 OP_1 的优先权低于 OP_2

$OP_1=OP_2$　　表示 OP_1 的优先权等于 OP_2

$OP_1>OP_2$　　表示 OP_1 的优先权高于 OP_2

表 4-1 定义了算符之间的这种优先关系。

表 4-1　算符之间的优先关系

OP_1 \ OP_2	+	−	*	/	(	)	#
+	>	>	<	<	<	>	>
−	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

具体实现时，在表达式的开始和结束之处各设置一个井号“#”，构成整个表达式的一对括号。根据运算规则 2，OP_1 为+、−、*、/

时，$OP_1<($；$OP_1)$)。例+<(。根据运算规则3，OP_1 和 OP_2 同时为+、一或同时为 *、/时，$OP_1>OP_2$。例+>+；->+；/>/；*>/等。在表中，(=)表示左右括号相遇，括号内的运算已经完成；同理，#=#表示整个算术表达式运算结束。)与(、#与)以及(与#之间无优先权关系，表示在算术表达式中不允许它们相继出现。如果出现这种情况，则表示这个表达式不正确。在下面讨论中，假设算术表达式的形式是正确的。

编译程序对算术表达式使用算符优先算法进行翻译计算时，需要设立两个工作栈：一个称为算符栈(OPTR)，用以存储算符(包括"#")；另一个称为操作数栈(OPND)，用于存储操作数和中间结果。算符优先算法的框图描述如图 4-12 所示。

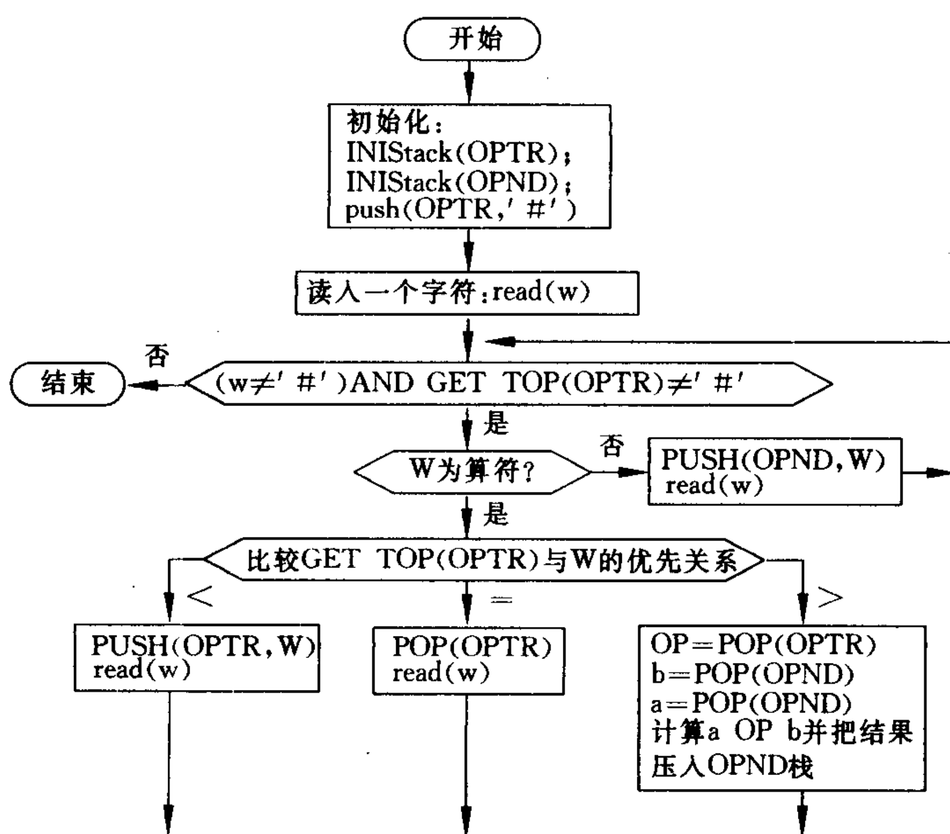


图 4-12 算符优先算法的框图

例如：用上述算法对算术表达式 6 *(3+2)的计算过程如下所示。

算符栈(OPTR)	操作数栈(OPND)	输入串
#		6*(3+2)#
#	6	*(3+2)#
# *	6	(3+2)#
# * (	6	3+2)#
# * (	6 3	+2)#
# * (+	6 3	2)#
# * (+	6 3 2	)#
# * (	6 5	)#
# *	6 5	#
#	30	#

习　　题

1. 试用栈编写一个判别表达式中开、闭圆括号是否配对出现的算法。
2. 设以带头结点的循环链表存储队列，并只设一个指针指向队尾元素结点，试编写相应的入队和出队算法。
3. 利用两个栈 s_1 和 s_2 来模拟一个队列，试编写利用栈的操作实现入队和出队算法。

第五日　串

前面我们在讨论线性表的操作时，都是对一个元素进行的。但在实际应用中，我们有时需要对一串元素进行操作。例如，在一个文字处理系统中，我们往往需要插入、删除、移动、替换一段文字，这就需要讨论一串字符的存储和处理的问题。

5.1　串的定义及其操作

一、串的基本概念

串又称字符串，是由零个或多个字符组成的有限序列，一般记为：

$$S='a_1,a_2\cdots,a_n' \quad (n\geqslant 0)$$

其中，S 为串的名；用成对单引号括起来的字符序列是串的值，而成对的单引号本身仅是作为串值的标记，不包含在串值中。$a_i(1\leqslant i\leqslant n)$通常是程序设计语言中允许使用的字符(包括字母、数字以及其它字符)，串中字符的数目 n 称为串的长度，长度为 0 的串称为空串。可以把串看作是以字符为数据元素的线性表。

串中任意多个连续的字符组成的子序列称为该串的子串，包含子串的串相应地称为主串。通常称字符在序列中的序号为该字符在串中的位置；子串在主串中的位置则以子串的第一个字符在主串中的位置来表示。

例如，假设 s，t，u，v 为如下的四个串：

s= 'Shang'；t= 'hai'；u= 'Shanghai'；v= 'Shang hai'

则它们的长度分别为 5,3,8,9;并且 s 为 u 和 v 的子串,其在 u、v 中的位置均为 1;t 也是 u、v 的子串,其在 u 中的位置为 6,在 v 中的位置为 7。当两个串长度相等且对应字符都相等时,称这两个串是相等的。在实际应用中,我们还会遇到一种串称之为空白串,它是由一个或多个空格(空白符)组成的串。要注意空串和空白串的区别,空串不含任何字符而空白串含有空白符。

二、串的基本操作

对串的操作有不同于其它线性结构的特点,它往往是对一组连续的字符进行。下面介绍一些有关串的常用的基本操作,其中 s、t 为串名。

1. 赋值操作(assign(s,t)):将 t 串的值赋给 s 串。
2. 判相等函数(equal(s,t)):若 s 串和 t 串相等,则返回函数值 TRUE;否则返回函数值 FALSE。
3. 连接函数(concat(s,t)):结果是把 t 串接在 s 串的后面构成一个新串。例如:s= 'Shang ',t= 'hai' 则 concat(s,t)= 'Shang hai'
4. 求长度(length(s)):其函数值为 s 串中字符的个数。
5. 求子串 substr((s,start,len)):若 1≤start≤length(s)+1 且 0≤len≤length(s)-start+1,则返回函数值为 s 串中从第 start 个字符起长度为 len 的字符串序列;否则返回一个特殊的串常量以表示所求子串不存在。
6. 定位函数(index(s,t)):若在主串 s 中存在和 t 串相等的子串,则返回函数值为在 s 串中第一个这样的子串在主串 s 中的位置;否则函数值为零。注意:在定位函数中 t 串不能为空串。
7. 置换函数(replace(s,t,v)):对字符串 s 中出现的所有子串 t,用字符串 v 进行替换,其中 t 串不能为空串。

   例如 s= 'ABCMDABC'、t= 'ABC'、v= 'H',则 replace(s,t,v)= 'HMDH'

8. 插入操作(insert(s,i,t)):在字符串 s 中位置 i 处插入字符串 t,要求 1≤i≤length(s)+1。

例如,s= 'ABCDEF'、t= 'xy',则 insert(s,3,t)= 'ABxyCDEF'

9. 删除操作(delete(s,i,len)):在字符串 s 中删除从位置 i 起长度为 len 的子串,要求 1≤i≤length(s)且 0≤len≤length(s)-i+1。

例如 s= 'ABCDEF',则 delete(s,3,3)= 'ABF'

5.2 串的存储结构

把字符串作为一种操作对象,它和其他的操作对象如整型量、实型量一样,在程序运行过程中,它的值是可以变化的,并且对它也要赋于一个名字,通过字符串的名字存取它的值。但是整型量、实型量在一种机器中都有固定的字长,而字符串由于串的长度不等,所需的存储空间也就不一样,存储结构当然要复杂一些。在这里我们对串的存储结构作简单的讨论。

一、串的顺序存储结构

一个字符串由一字符序列组成,而每个字符所需的存储空间是一定的,所以我们可以用一个字符数组来存放一个字符串。这样可以通过串名(也就是数组名)直接访问到串值。在操作时一个字符数组作为一个整体参加,它代表了一个字符串。下面用 PASCAL 语言中的一维数组类型定义字符串:

```
CONST maxlen={允许字符串的最大长度};
TYPE strtp=RECORD
                ch:ARRAY[1..maxlen] OF char; curlen:0..maxlen
              END;
```

其中,ch 为存储串值的一维数组,它的每个分量存放一个字符;curlen 指示串的当前长度。显然,在这种存储结构下字符串的长度不

能超过 maxlen。

二、串的链式存储结构

对于字符串，也可以采用类似于线性表的链式存储结构。例如对于字符串 'This is a string' 可以用如图 5-1 所示的存储结构。在用链表存放字符串时，一个结点中可以存放一个字符，如图 5-1(a)所示；也可以考虑存放多个字符，如图 5-2(b)所示一个结点存放 4 个字符。当结点中存放的字符数目大于 1 时，由于串长不一定正好是结点大小的整数倍，因此链表中的最后一个结点不一定会被串值占满，此时通常用一个不属于串的字符集的特殊字符(如井号"#")补上。

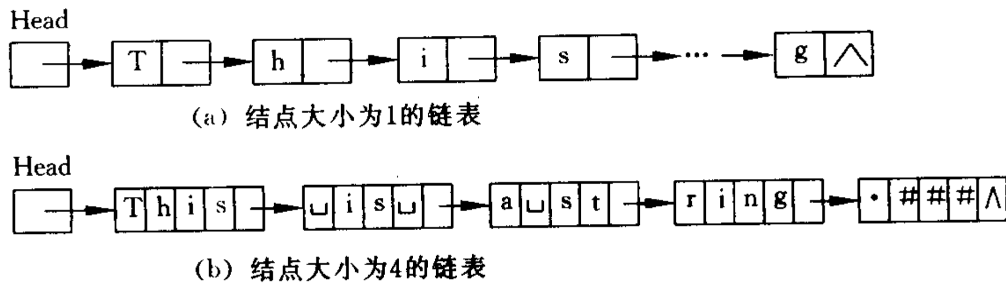


图 5-1　串的链式存储结构

为了便于操作，当用链表存储串值时，除设置指示第一个结点的首指针外还可以附设一个尾指针，指向链表中最后一个结点，并记录当前串的长度。下面我们用 PASCAL 语言定义字符串的链式存储结构。

```
CONST chunksize={一个结点中存放的字符数}；
TYPE pointer= ↑charnode;
     charnode=RECORD
                ch:ARRAY[1..chunksize] OF char;
                next:pointer;
              END;
     linkstrtp=RECORD
                head,tail:pointer;
                length:integer;
              END;
```

在链式存储结构中，结点大小的选择是很重要的，它直接影响着串的处理效率。在这里我们定义串的存储密度为：

存储密度＝串值所占的存储单元/实际分配的存储单元

显然存储密度越小(如结点大小为 1)处理越方便，然而存储占用量越大。特别是如果在串的处理过程中需要经常进行内存和外存交换数据的话，则会因为存储密度小，而使内存和外存交换操作过多，影响处理的总效率。

三、堆结构

从上述两种存储结构的讨论可知，无论是采用顺序存储结构还是链式存储结构，它们都存在一些弊病。当用顺序存储结构存储串值时，由于在串的类型定义中必须预先规定串值允许的最大长度，而在一般情况下，串的长度变化范围较大，所以当多数串的长度较短时空间的利用率很低；而另外则由于限定了串的最大长度，使串的某些操作如：连接、置换等受到长度的限制。当用链式存储结构时，虽然链表的结构比较灵活，使串的长度不受限制，但却受到存储密度的制约，如果提高了存储密度，必然使串的操作复杂化。

为此，在很多实际应用的串处理系统中，对串采用一种动态的存储结构称为堆结构。它就是在系统中开辟一个容量很大、地址连续的存储空间作为存放串值的可利用空间。当建立一个新串时，系统就从可利用空间中分配一个大小和串的长度相同的、地址连续的存储空间用于存储新串的值。这样所有串的串值都存储在这个可利用空间中。同时为每个串建立一个索引，以记录该串的长度以及其串值在可利用空间中的起始位置。

假设以一维数组表示存储串值的可利用空间：

```
store : ARRAY [1..maxsize] OF char;
```

其中，maxsize 表示可利用空间的最大容量；并设一个指针 free(一个整型变量，初值为 1)，用于指示可利用空间中尚未进行分配的空间的起始地址。

串索引的 PASCAL 描述如下：

```
TYPE stringtype=RECORD
          length,stadr:integer
        END;
```

其中 length 域存放串的长度，stadr 域指示串值序列在 store 中的起始位置。借助于这两个域可在串名和串值之间建立起一个对应关系，称做串名的存储映象。例如，图 5-2 所示为四个串 a、b、c、d 的存储映象以及可利用空间的状态。其中，a 串的值为 'BEI'，长度为 3，起始地址为 1；b 串的值为 ' JING'，长度为 5，起始地址为 4；c 为空串；d 串的值为 'SHANGHAI'，长度为 8，起始地址为 9；可利用空间中尚未被分配的区间的起始地址为 17(free=17)。

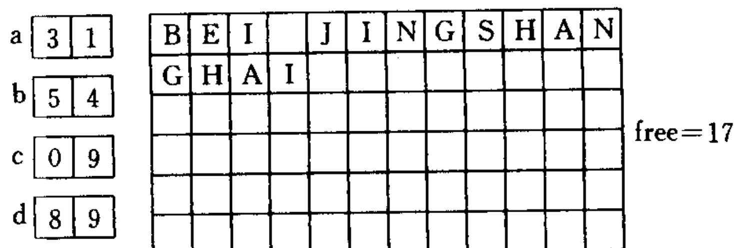


图 5-2　串的存储映象示例

5.3　串基本操作的实现

在这一节中，我们将讨论串的基本操作的实现；在不同的存储结构上，其实现的算法是不同的。在这里我将讨论在顺序存储结构上串操作的实现，串的类型定义为上一节中的 strtp。

一、赋值运算 assign(s,t)

该操作就是要将 t 串的值赋给 s 串。其算法的 PASCAL 语言描述如下：

```
PROCEDURE assign(VAR s:strtp;t:strtp);
VAR i:integer;
BEGIN
  FOR i:=1 TO t.curlen DO
```

```
    s.ch[i]:=t.ch[i];
    s.curlen:=t.curlen
  END;
```

二、判相等函数 equal(s,t)

在这里我们可以先判别串 s 和 t 的长度是否一样，如果一样长，再判串值是否一样。算法的 PASCAL 语言描述如下：

```
FUNCTION equal(s,t:strtp):boolean;
VAR b:bolean;
    i:integer;
BEGIN
  IF s.curlen=t.curlen THEN
    BEGIN
      b:=TRUE;i:=1;
      WHILE (i<=s.curlen) AND b DO
        BEGIN
          IF s.ch[i]<>t.ch[i]
            THEN b:=FALSE;
          i:=i+1
        END;
      return(b)
    END
    ELSE
      return(FALSE)
END;
```

三、串的连接 concat(s,t)

两个串的连接就是将串 t 紧接在另一个串 s 的后面，组合成一个新串。由于串值采用顺序存储结构存放，所以连接后的串长可能会超过串长允许的长度，这就要给出一个“溢出”信息，图 5-3 为串连接算法的框图描述。

该算法的 PASCAL 语言描述如下：

```
FUNCTION concat(VAR s:strpt; t:strpt):boolean;
VAR p,i:integer;
BEGIN
  IF (s.curlen+t.curlen)>maxlen
    THEN return(FALSE)
    ELSE
      BEGIN
        p:=s.curlen;
        FOR i:=1 TO t.curlen DO
          s.ch[p+i]:=t.ch[i];
        s.curlen:=s.curlen+t.curlen;
        return(TRUE)
      END
END;
```

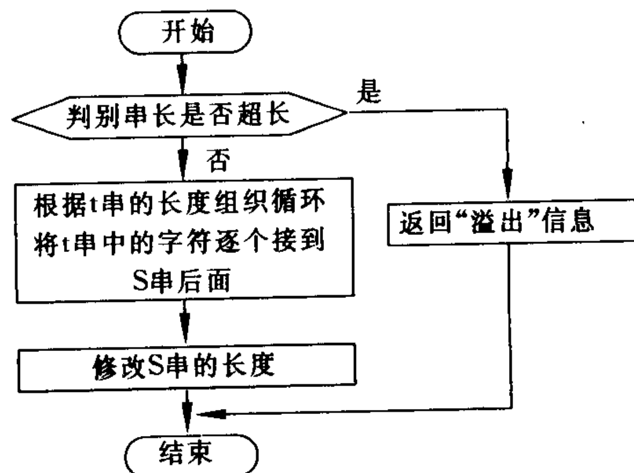


图 5-3　串的连接算法框图

四、求子串 substr(s,start,len)

求子串的过程也是复制字符序列的过程，但当所求子串在 s 串中的起始位置或子串长度不合理时，应给出“出错”信息。该算法的 PASCAL 语言描述如下：

```
FUNCTION stubstr(VAR sub:srtp; s:strtp;
                 start,len:integer):boolean;
```

```
VAR p,i:integer;
BEGIN
  IF (1<=start) AND (start<=s.curlen+1) AND
   (0<=len) AND (len<=s.curlen-start+1)
   THEN BEGIN
          p:=start-1;
          FOR i:=1 TO len DO
            sub.ch[i]:=s.ch[p+i];
          sub.curlen:=len;
          return(TRUE)
         END
   ELSE BEGIN
         sub.curlen:=-1;
         return(FALSE)
        END;
END;
```

下面我们举例说明此函数的调用情况。

例如:s= 'This is a string. '则

1. b:=substr(t,s,11,6);得到子串 t= 'string'、b 为 TRUE。
2. b:=substr(t,s 11,8);得到子串 t 的值不确定、b 为 FALSE。

五、定位函数 index(s,t)

子串的定位操作就是求子串在主串中的位置,通常称为串的模式匹配(其中子串 t 称为模式)。这个操作是各种串处理系统中最重要的操作之一。其算法的基本思想是:从主串 s 的第一个字符起和模式 t 的第一个字符进行比较,如果相等则继续逐个比较后续字符;否则从主串的第二个字符起再重新和模式的第一个字符进行比较。依次类推,直至模式 t 中的每个字符依次和主串 s 中从第 i 个字符开始的一个子串相等,则称模式匹配成功,函数返回模式 t 的第一个字符在主串 s 中的序号;否则称匹配不成功,函数返回零。如下展示了模式 t= 'abcac'和主串 s= 'ababcabcacbab'的匹配过程。

```
     ↓ i=3
ababcabcacbab          第一趟匹配
abc
     ↑ j=3
   ↓ i=2
ababcabcacbab          第二趟匹配
 a
 ↑ j=1
        ↓ i=7
ababcabcacbab          第三趟匹配
  abcac
        ↑ j=5
   ↓ i=4
ababcabcacbab          第四趟匹配
   a
   ↑ j=1
    ↓ i=5
ababcabcacbab          第五趟匹配
    a
    ↑ j=1
          ↓ i=11
ababcabcacbab          第六趟匹配
     abcac
          ↑ j=6
```

结果，函数返回模式 t 在主串 s 中的序号 6(i-t.curlen)。该算法的 Pascal 语言描述如下。

```
FUNCTION index(s,t:strtp):integer;
VAR i,j:integer;
BEGIN
  i:=1; j:=1;
```

```
  WHILE (i<=s.curlen) AND (j<=t.curlen) DO
    IF s.ch[i]=t.ch[i]
      THEN BEGIN i:=i+1; j:=j+1 END
      ELSE BEGIN i:=i-j+2; j:=1 END;
    IF j>t.curlen
      THEN return(i-t.curlen)
      ELSE return(0)
END;
```

5.4 文本编辑

文本编辑是字符串处理中最常见的应用之一。文本编辑的含义是:删除或替换文本中指定的一些行或行中的某些字符;插入一些新的行或字符。计算机高级语言(如PASCAL语言)源程序的修改以及报纸、书籍、期刊、文章的修改,都可以在计算机上通过文本编辑程序用一系列编辑命令完成。例如:一份用PASCAL语言编写的源程序,可以看成是一个文本。它不是简单的由一个长字符串组成,一般要分成许多页,每页由若干行组成。表示这种行页结构的方法,在不同的计算机系统中是不同的,常用的方法是以特定的控制字符来实现。如用ASCII字符集中的回车符CR(用符号"←"表示)和换行符(用符号"↓"表示)表示一行结束;用换页符FF(用符号"▬"表示)表示一页结束。当用户调用文本编辑程序时,在将文本输入到内存的同时,由编辑程序建立一个行表,行表中的每一项表示一行,它由行号、存储该行字符串的起始地址以及该行字符串的长度等组成。在文本编辑过程中,编辑程序对文本的访问是以行表的内容为依据的。所以对文本的修改就直接反映在行表上。例如有下面一段程序需输入到内存,其中符号"Φ"表示空格符。

```
PROCEDUREΦEX;←↓
ΦVARΦi,j:integer;←↓
ΦBEGIN←↓
ΦΦread(i);←↓
```

·ΦΦj:=i+i*i;←↓

ΦEND;←↓

由于输入时发生了差错，实际输入到内存中的的文本如图 5-4 所示。假定存储地址从 201 开始，则编辑程序建立的相应行表如图 5-5(a)所示。

201															216
P	R	O	C	E	D	U	R	E	Φ	E	X	;	←	↓	Φ
							224								232
B	E	G	I	N	←	↓	Φ	B	E	G	I	N	←	↓	Φ
											244				
Φ	r	e	a	d	(	i	)	;	←	↓	Φ	Φ	j	:	=
								257							264
i	+	i	*	i	;	←	↓	Φ	E	N	D	;	←	↓	

图 5-4　错误文本的存储状态

对这个有错的文本，其修改步骤如下：

1. 删除第 110 行。这只要在行表中删除 110 行，此时由于不能访问到地址为 216 至 223 的存储空间，就等于删除了第 110 行。

2. 插入新的一行，其内容为："ΦVARΦi,j:integer;←↓"。这时，可根据新行的长度，将其存放在可利用空间的空闲区，即可从地址 264 开始存储此行，而其行号仍为 110。这样修改后，新的正确的行表内容如图 5-5(b)所示。经编辑修改后的正确文本在可利用空间中的状态如图 5-6 所示。

行号	起始地址	串长
100	201	15
110	216	8
120	224	8
130	232	12
140	244	13
150	257	7

(a)错误文本的行表

行号	起始地址	串长
100	201	15
110	264	19
120	224	8
130	232	12
140	244	13
150	257	7

(b)正确文本的行表

图 5-5　行表示例

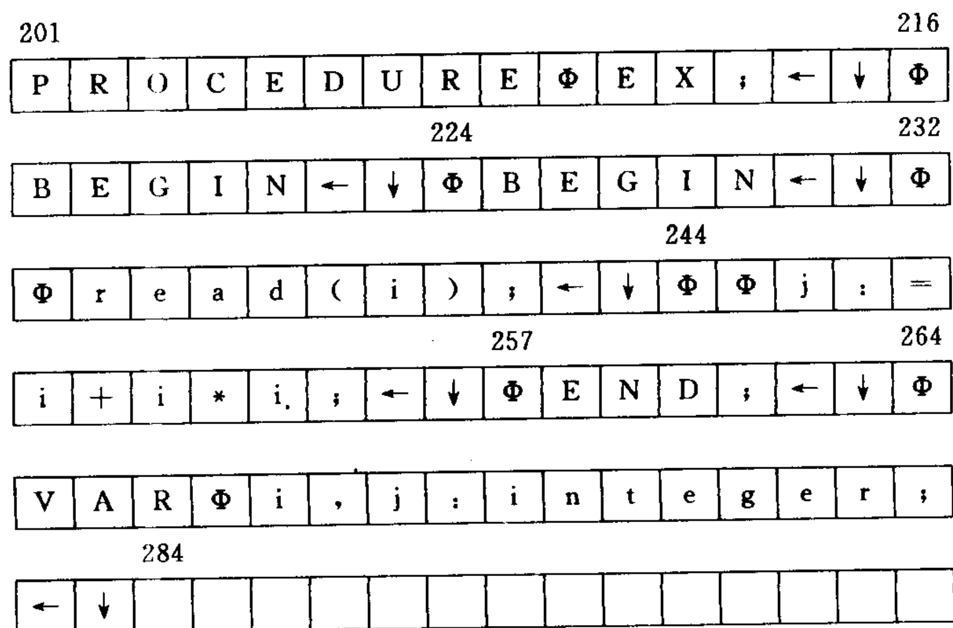


图 5-6 正确文本的存储状态

在上例中，插入的新行，实际存储在原文本的后面。所以，它没有移动数据。当然，也可以通过移动数据，把新行插在第 100 行的后面。这些具体的算法，读者可利用串的基本运算自己完成。

习　题

1. 设 S='I AM A STUDENT'，T='GOOD'，Q='WORKER'。求：
   length(S)，length(T)，substr(S,8,7)，substr(T,2,1)，index(S, 'A')，replace(S,'STUDENT',Q)，concat(substr(S,6,2),concat(T,substr(S,7,8)))。
2. 设 S='(ABC)+ *'，T='(A+C) * C'。试利用联接、求子串和置换等基本运算，将 S 变为 T。
3. 给定两个串 a 和 b，求在 a 串中第一次出现，而在 b 串中不出现的字符的序号。试编写此算法。

第六日　数　　组

数组是大家所熟悉的一种数据类型，几乎所有的程序设计语言都设定数组类型为固有类型。所以在本日中仅简单地讨论数组的逻辑结构定义及其存储结构。

6.1　数组的定义和运算

数组可以看成是线性表的推广，数组中每个元素是由一个值和一组下标组成的。如图 6-1 所示，就是一个二维数组，记作 A[1..m, 1..n]。二维数组也称为矩阵。

$$A=\begin{bmatrix} a_{11} & a_{12}\cdots & a_{1n} \\ a_{21} & a_{22}\cdots & a_{2n} \\ \vdots & \vdots & \vdots \\ a_{m1} & a_{m2}\cdots & a_{mn} \end{bmatrix}$$

图 6-1　二维数组示例

在这个二维数组中，每个元素 a_{ij}(在 PASCAL 语言中记为 a[i,j])都同时属于两个线性表，一个是第 i 行的行表($a_{i1}, a_{i2}, \cdots, a_{in}$)；另一个是第 j 列的列表($a_{1j}, a_{2j}, \cdots, a_{mj}$)。这种行表和列表都相当于一维数组。我们可以把二维数组看成是这样的一个线性表，它的每个数据元素是一个线性表。例如图 6-1 所示的二维数组，它可以看成是一个线性表：

$$A=(\alpha_1, \alpha_2, \cdots, \alpha_n)$$

其中每一个元 $\alpha_j(1\leqslant j\leqslant n)$是一个列向量的线性表：

$$\alpha_j=(a_{1j}, a_{2j}\cdots, a_{mj})$$

当然，我们也可以把这个二维数组看成是如下一个线性表：

$$A=(\beta_1, \beta_2\cdots, \beta_m)$$

其中每一个元 $\beta_i(1\leqslant i\leqslant m)$是一个行向量的线性表：

$$\beta_i=(a_{i1},a_{i2}\cdots a_{in})$$

同样，三维数组可以看成数据元素为二维数组的线性表。依次类推，n 维数组可以看成数据元素为 n－1 维数组的线性表。因此，数组是线性表的一种推广。在这里值得大家注意的一点就是一个数组中所有的数据元素都必须属于同一数据类型。对于数组通常只有两种基本操作：

1. 给定一组下标，存取相应的数据元素。
2. 给定一组下标，修改相应数据元素中的数据项。

6.2 数组的顺序存储结构

数组的顺序存储结构就是用一组连续的存储单元顺序存放数组元素。由于存储单元是一维结构，而数组是一个多维结构，所以用一组连续的存储单元存放数组元素时就要考虑存放的次序问题。

首先我们讨论二维数组，元素间的次序可以有两种排序方法，一种方法就是按行的次序进行排列，称为“行优先序”，它就是把数组元素按行表次序、第 i+1 行的元素紧跟在第 i 行元素后面进行存储，如图 6-2(b)所示；另一种方法就是按列的次序进行排列，称为“列优先序”，它就是把数组元素按列表次序、第 j+1 列元素紧跟在第 j 列元素后面进行存储，如图 6-2(c)所示。

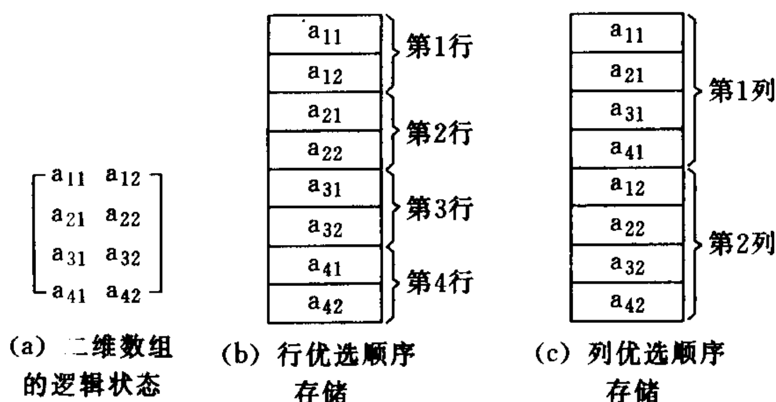


图 6-2 二维数组的顺序存储结构

同样，对于 n 维数组也有上述两种不同的顺序存储方法：“右下

标优先序”和“左下标优先序”，它们分别相当于二维数组的“行优先序”和“列优先序”。把 n 维数组的元素按上述方式顺序存放在存储单元中，则每个元素的存储地址可以用一条公式计算出来，这条公式称为“地址公式”。假设每个数据元素只占一个存储单元并以“右下标优先序”进行顺序存储，则“地址公式”为：

对于一维数组 A[1..m]，其中任一元素 $a_i(1 \leqslant i \leqslant m)$ 的存储地址为：

$$LOC(a_i)=LOC(a_1)+(i-1)$$

其中，$LOC(a_i)$ 表示元素 a_i 的存储地址；$LOC(a_1)$ 为元素 a_1 的存储地址，即一维数组 A 的起始存储位置，又称基地址。

对于二维数组 A[1..m,1..n]，其中任一元素 a_{ij} 的存储地址为：

$$LOC(a_{ij})=LOC(a_{11})+n*(i-1)+(j-1)$$

其中，$LOC(a_{ij})$ 表示元素 a_{ij} 的存储地址；$LOC(a_{11})$ 为元素 a_{11} 的存储地址，它是二维数组 A 的基地址。图 6-3 是这个地址公式的示意图。

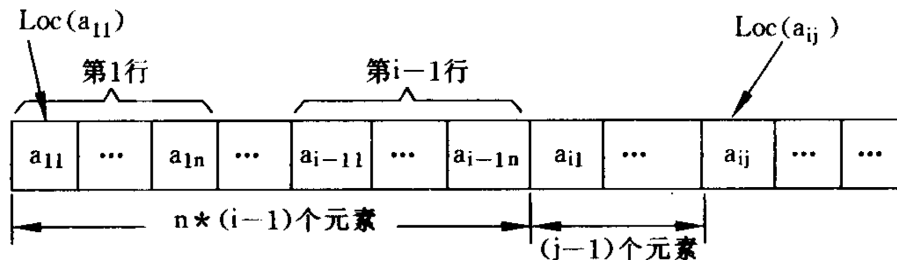


图 6-3 二维数组 A[1..m,1..n]的存储状态

对于三维数组 A[1..l,1..m,1..n]，可以分解为 l 个 m*n 的二维数组，则其中任一元素 a_{ijk} 的存储地址为：

$$LOC(a_{ijk})=LOC(a_{111})+m*n*(i-1)+n*(j-1)+(k-1)$$

其中，$LOC(a_{111})$ 为三维数组 A 的基地址，图 6-4 为三维数组的地址公式示意图。

从上述地址公式，可以很容易地推广到 n 维数组，设有 n 维数组 A[$1..d_1,1..d_2,\cdots,1..d_n$]，则其中任一元素 $a_{i_1i_2\cdots i_n}$ 的存储地址为：

$$\begin{aligned}LOC(a_{i_1i_2\cdots i_n})=LOC(a_{11\cdots 1})&+d_2*d_3*\cdots*d_n*(i_1-1)\\&+d_3*d_4*\cdots*d_n*(i_2-1)\\&+\cdots+d_n*(i_{n-1}-1)+(i_n-1)\end{aligned}$$

即：

$$LOC(a_{i1i2\cdots in})=LOC(a_{11\cdots 1})+\sum_{j=1}^{n}S_j*(i_j-1)$$

其中，$S_j=\prod_{k=j+1}^{n}d_k$，　$1\leqslant j<n$；且　$S_n=1$

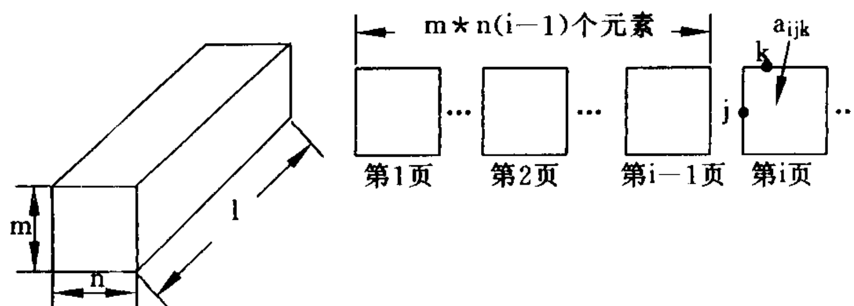


图 6-4　三维数组 A[1..l,1..m,1..n]的存储状态

从中可以看出，在顺序存储结构中，数组元素的存储位置是其下标的一个线性函数，所以存取数组中任一元素的时间是相等的，我们称具有这一特点的存储结构为随机存储结构。

在上述公式中，n 维数组中每一维的下界都定为 1，并且每个元素只占一个存储单元；如果下界不为 1，每个元素占 c 个存储单元，则应如何修改数组的地址公式？还有如果数组元素以“左下标优先序”顺序存储，则地址公式又是如何？请读者根据上述讨论自已思考。

6.3　矩阵的压缩存储

在很多科学和工程计算中，矩阵是研究的主要数学对象之一。在这里，我们不是讨论矩阵的本身，而是要讨论如何存储矩阵中的元素，从而使矩阵的各种运算能有效地进行。

一般情况下，用高级语言编制程序时，都是用二维数组的顺序存储结构存放矩阵元素。然而，在数值分析中经常出现一些阶数很高的矩阵，并且在矩阵中有许多值相同的元素或者值为零的元素（零元素）。有时为了节省存储空间，需要对这些矩阵进行压缩存储。所谓压缩存储是指：为多个值相同的元素只分配一个存储空间；对零元素

不分配空间。如果这些值相同的元素或零元素在矩阵中的分布有一定规律，则我们称这类矩阵为特殊矩阵；反之，称为稀疏矩阵。下面分别对这两类矩阵讨论它们的压缩存储。

一、特殊矩阵

如果在一个 n＊n 阶的矩阵 A 中，其元素满足下述性质：

$$a_{ij}=a_{ji} \qquad 1<=i、j<=n$$

则称 A 为 n＊n 阶对称矩阵。

对于对称矩阵，我们可以为每一对对称元素(a_{ij},a_{ji})分配一个存储空间，这样 n^2 个元素就可以压缩存储到 n(n+1)/2 个存储空间中。不失一般性，对这样的矩阵以行优先序，把它的下三角(包括对角线)元素，存放在一维数组 Sa[1..(n+1) /2]中，则一维数组中元素 Sa[k]和矩阵中元素 a_{ij}之间存在如下一一对应的关系：

$$K=\begin{cases} i(i-1)/2+j & 当\ i>=j \\ j(j-1)/2+i & 当\ i<=j \end{cases}$$

由此可见，对于任意给定的一组下标(i,j)，均可在 Sa 中找到矩阵元 a_{ij}；反之，对所有的 k=1,2,…,n(n+1)/2 都能确定 Sa[k]中的元素在矩阵中的位置(i,j)。这种一维数组 Sa 称为 n＊n 阶对称矩阵 A 的压缩存储，图 6-5 为对称矩阵压缩存储的示意图。

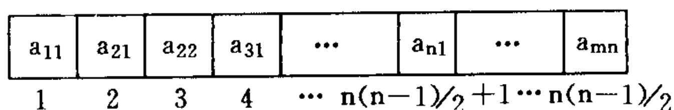


图 6-5　对称矩阵的压缩存储

对称矩阵的这种压缩存储方法也适用于下(上)三角矩阵。所谓下(上)三角矩阵是指矩阵的上(下)三角(不包括对角线)中的元素均为常数 c 或零。只是对于下(上) 三角矩阵，除了只存储其下(上)三角中的元素外，还必须再加一个存储单元，用于存储常数 c。

在这些特殊矩阵中，非零元素的分布都有一个明显的规律，从而可以将其压缩存储到一维数组中。然而，在实际应用中还会经常遇到这样一类矩阵，其非零元素很少而且分布没有规律，我们称之为稀疏

矩阵。如图 6-6(a)所示,M 为 6 * 7 的矩阵,共有 42 个元素,其中只有 8 个非零元素,则 M 就是一个稀疏矩阵。

$$M=\begin{bmatrix} 0 & 8 & 0 & -3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ -12 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 15 \\ 0 & 6 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(a) 稀疏矩阵M

$$N=\begin{bmatrix} 0 & 0 & 0 & -12 & 0 & 0 \\ 8 & 0 & 0 & 0 & 0 & 6 \\ 0 & 0 & 24 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 15 & 0 \end{bmatrix}$$

(b) 稀疏矩阵N

图 6-6 稀疏矩阵图示

由于这类矩阵中非零元素的分布没有规律,所以压缩存储就要比特殊矩阵复杂。下面我们讨论用三元组和十字链表对稀疏矩阵进行压缩存储。

二、稀疏矩阵的三元组存储结构

我们知道,矩阵中的每一个元素都对应有一组下标(i,j)和元素本身的值;每当存储一个矩阵元素,就要考虑如何表示这组下标和元素本身的值。在前面讨论的无论是数组的顺序存储结构,还是特殊矩阵的压缩存储都是通过存储地址隐式地表示了元素的下标,而元素的值则是显式地存放在存储单元中。将稀疏矩阵进行压缩存储,显然要把每个非零元素的下标和值都显式地表示出来;我们可以考虑用一个三元组(i,j,val)表示,其中 i 和 j 分别表示元素所在的行和列,val 表示元素的值。例如,对图 6-6(a)所示的稀疏矩阵,M 中的八个非零元素可以用如下八个三元组表示,它们可按行优先序排列:(1,2,8)、(1,4,-3)、(2,6,4)、(3,3,24)、(4,1,-12)、(4,5,2)、(5,7,15)、(6,2,6)。这些三元组组成一个线性表,线性表中的每个数据元素是一个三元组。另外,为了唯一地确定稀疏矩阵,在线性表的第零个位置上增加一个表示矩阵行数、列数,和非零元素个数的三元组。对这个线性表,可采用顺序存储结构。例如,对于图 6-6(a)所示的稀疏矩阵 M,其三元组形式表示的顺序存储结构如图 6-7(a)所示。

	行	列	
A[0]	6	7	8
A[1]	1	2	8
A[2]	1	4	−3
A[3]	2	6	4
A[4]	3	3	24
A[5]	4	1	−12
A[6]	4	5	2
A[7]	5	7	15
A[8]	6	2	6

(a) 矩阵M的三元组向量A

	行	列	
B[0]	7	6	8
B[1]	1	4	−12
B[2]	2	1	8
B[3]	2	6	6
B[4]	3	3	24
B[5]	4	1	−3
B[6]	5	4	2
B[7]	6	2	4
B[8]	7	5	15

(b) 矩阵N的三元组向量B

图 6-7 稀疏矩阵的三元组存储结构

其中，a[0]表示矩阵 M 共有 6 行、7 列、共 8 个非零元素；a[1]、a[2]、…、a[8]分别表示了矩阵 M 中的 8 个非零元素。

用三元组实现稀疏矩阵的压缩存储，那么如何在三元组上进行矩阵的运算呢？下面以矩阵的转置运算为例进行讨论。

对于一个 $m*n$ 的矩阵 M，它的转置矩阵 N 是一个 $n*m$ 的矩阵，并且 $N[i,j]=M[j,i]$，$1\leqslant i\leqslant n$、$1\leqslant j\leqslant m$。例如，图 6-6(a)所示的矩阵 M，其转置矩阵 N 如图 6-6(b)所示。对于矩阵 N，其三元组表示如图 6-7(b)所示。当然我们关心的是如何从向量 a 中求得向量 b？对于向量 a 中的每一个分量来说，只要将其行域和列域的内容进行交换，就得到了向量 b 中的一个分量。向量 a 中的分量次序是以矩阵 M 的非零元素的“行优先序”进行排列的，对于矩阵 N 的三元组向量 b，其分量当然也应该按“行优先序”进行存储。然而，如果按向量 a 中的顺序将每个分量进行转换，显然转换后得到的向量 b 不符合“行优先序”。例如，依次把向量 a 中分量进行行、列交换，得到：

$$
\begin{array}{lcl}
(6,7,8) & & (7,6,\ 8) \\
(1,2,3) & & (2,1,\ 8) \\
(1,4,3) & \Rightarrow & (4,1,-3) \\
(2,6,4) & & (6,2,\ 4) \\
(3,3,4) & & (3,3,24) \\
\vdots & & \vdots
\end{array}
$$

要使转置矩阵 N 的三元组向量 b 以"行优先序"排列，容易想到的一个方法就是按向量 a 中的顺序进行转换，并且每转换完一个分量就按其在向量 b 中的"行优先序"把它插到适当的位置。这样做虽然能满足要求，但移动元素十分频繁。为了避免元素移动，可以采取以下二种方法。

1. 按矩阵 M 中的列序进行转置

这种方法就是：首先考虑把矩阵 M 的第 1 列非零元素转换成矩阵 N 的第 1 行元素；然后再考虑把矩阵 M 的第 2 列非零元素转换成矩阵 N 的第 2 行元素；依次类推，最后把矩阵 M 的第 n 列（最后一列）非零元素转换成矩阵 N 的第 n 行元素。当然，为了寻找矩阵 M 中每一列的非零元素，都需要对向量 a 全部扫描一遍，图 6-8 为其算法的框图描述。

根椐图 6-8 所示的算法框图，可用 PASCAL 语言描述如下：

```
CONST
  maxmum={允许的非零元素个数};
TYPE
  node=RECORD
         i,j:integer;
         val:integer;
       END;
  listar=ARRAY[0..maxnum] OF node;
PROCEDURE trans-sparmat(VAR b:listar; a:listar);
VAR
  col,k,p,n,t:integer;
BEGIN
  b[0].i:=a[0].j;
  b[0].j:=a[0].i;
  b[0].val:=a[0].val;
  n:=a[0].j; t:=a[0].val;
  IF t<>0 THEN
    BEGIN
      P:=1;
```

```
    FOR col:=1 TO n DO
      FOR k:=1 TO t DO
        IF a[k].j=col THEN
          BEGIN
            b[p].i:=a[k].j;
            b[p].j:=a[k].i;
            b[p].vol:=a[k].vol;
            p:=p+1
          END
  END
END;
```

在过程 trans-sparmt 中，为了依次在向量 b 中存入三元组，定义了一个指示器 p，用于指示向量 b 中按行优先序应存入的非零元素的当前位置，其初值为 1；当向量 b 中存入一个非零元素后，就修改一下 p(p:=p+1)。

此转置算法的时间主要花在两个 FOR 语句上，所需的时间复杂度为 O(n * t)。当稀疏矩阵的非零元素的数目 t 和 m * n 等数量级时，其时间复杂度为 O(n2 * m)。而在一般情况下，矩阵的转置算法为：

```
FOR col:=1 TO n DO
  FOR row:=1 TO m DO
    N[col,row]:=M[row,col];
```

其时间复杂度为 O(m * n)，由此可见 trans-sparmt 算法仅适用于 t<<n * m的情况。

2. 按矩阵 M 中的行序进行转置

上面介绍的方法是按矩阵 M 中的列序进行转置的。当然，转置方法也可以按矩阵 M 中的行序进行，但转置后的三元组需按矩阵 N 的行优先序直接填到向量 b 的适当位置上。这样按矩阵 M 的行序进行转置，则只需对向量 a 扫描一次；而按矩阵 N 的行优先序直接将三元组填到向量 b 的适当位置，可以避免向量 b 中元素的移动。显然，这种方法是十分可取的。但问题是如何确定矩阵 M 中的一个非

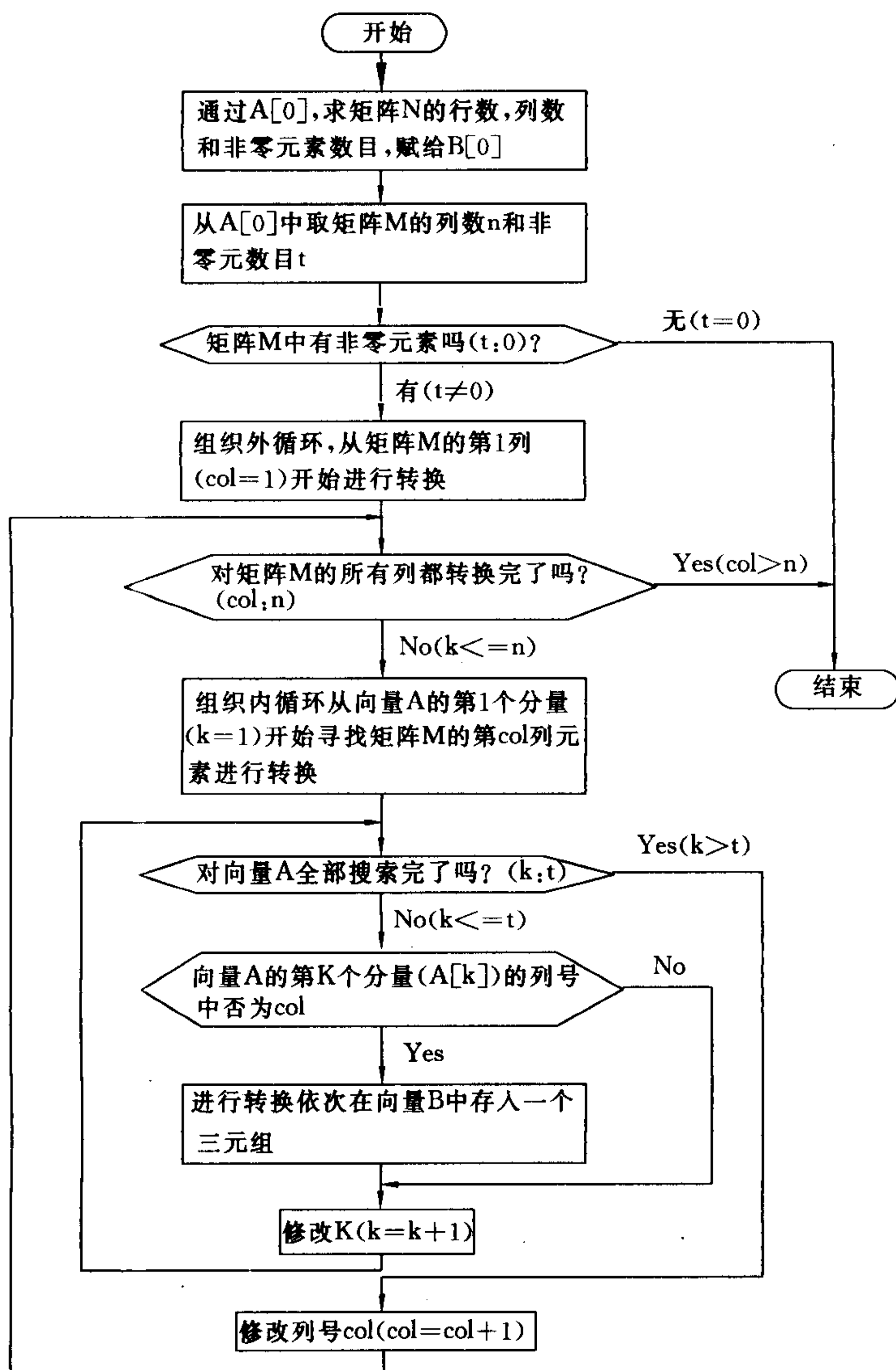


图 6-8　按矩阵 M 的列序进行转置的算法框图

零元素在向量 b 中的位置？对于这个问题，我们可以这样做：首先求出矩阵 M 中每一列的非零元素的数目，存放在一个数组 num[1..n]

中(n 是矩阵 M 列数),这个数组的每一个分量 num[col](1≤col≤n)表示了矩阵 M 的第 col 列中非零元素的数目;然后,由于矩阵 M 中第 1 列的第一个非零元素转置后总是为矩阵 N 的第 1 行的第一个非零元素,所以,它必然存放在 b[1]中。这样,根据数组 num[1..n]就可以推算出矩阵 M 中每一列的第一个非零元素在向量 b 中的位置;我们用一个数组 pot[1..n]来记录这些位置,这个数组的每个分量 pot[col]表示了矩阵 M 中第 col 列的第一个非零元素在向量 b 中的位置,并有:

pot[1]:=1;

pot[col]=pot[col-1]+num[col-1], 2<=col<=n

例如,对图 6-6(a)所示的矩阵 M,其 num 和 pot 数组的值如下:

col	1	2	3	4	5	6	7
num[col]	1	2	1	1	1	1	1
pot[col]	1	2	4	5	6	7	8

有了数组 pot 就知道了矩阵 M 中每一列的第一个非零元素在向量 b 中的位置,这样我们就可以按矩阵 M 的行序进行转置。当按行序把矩阵 M 中第 col 列的一个非零元素转置时,就把转置后的三元组存放在向量 b 中由 pot[col]所指示的位置,然后修改 pot[col](pot[col]:=pot[col]+1)使其指向矩阵 M 中第 col 列的下一个非零元素在向量 b 中的位置。这样,我们依靠数组 pot,就可实现矩阵的转置。其算法的框图如图 6-9 所示。

根据这个框图,我们用如下的 PASCAL 语言描述这个算法。

```
CONST
  maxn={矩阵中最大的列数};
PROCEDURE fast-trans(VAR b:listar; a:listar);
VAR
  pot,num:ARRAY[1..maxn] OF integer;
  col,k,n,t:integer;
BEGIN
  b[0].i:=a[0].j;
  b[0].j:=a[0].i;
```

```
b[0].val:=a[0].val;
n:=a[0].j; t:=a[0].val;
IF t<>0 THEN
  BEGIN
    FOR col:=1 TO n DO
      num[col]:=0;
    FOR k:=1 TO t DO
      num[a[k].j]:=num[a[k].j]+1;
    pot[1]:=1
    FOR col:=2 TO n DO
      pot[col]:=pot[col-1]+num[col-1];
    FOR k:=1 TO t DO
      BEGIN
        col:=a[k].j;
        b[pot[col]].i:=a[k].j
        b[pot[col]].j:=a[k].i;
        b[pot[col]].val:=a[k].val;
        pot[col]:=pot[col]+1
      END
  END
END;
```

显然，这个算法比上一个算法多用了两个辅助向量(num、pot)。从时间上看，算法中有四个并列的单循环，循环次数分别为 n 和 t，所以总的时间复杂度为 $O(n+t)$；当 $t \sim m*n$ 时，为 $O(m*n)$，与通常算法的时间复杂度相同；当 $t << m*n$ 时，算法明显是高效的。

当然，要在这种三元组形式的顺序存储结构中插入或删除一个元素，就要移动元素，这就比较麻烦了。如果要在这种存储结构上施行矩阵的加法、乘法等操作，由于运算过程中非零元素经常发生变化，所以不宜用三元组形式的顺序存储结构。这时，宁可采用占内存空间较多的二维数组顺序存储结构，或者采用下面介绍的十字链表。

三、稀疏矩阵的十字链表存储结构

稀疏矩阵的十字链表存储结构是用多重链表来存储矩阵，它实

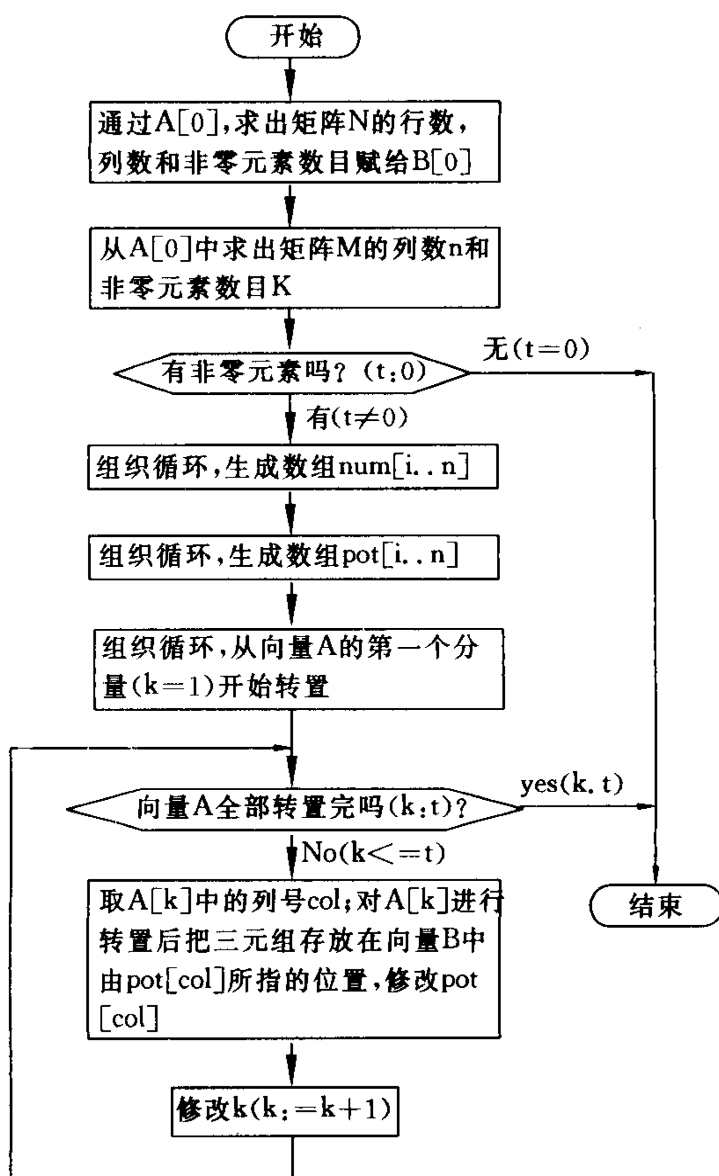
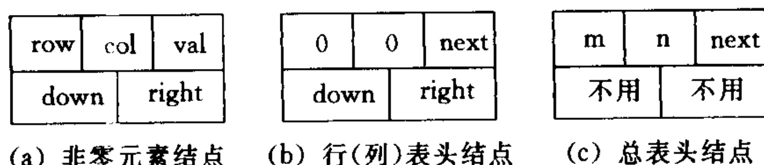


图 6-9 按矩阵 M 的行序进行转置的算法框图

际上是三元组形式的线性表的链式存储结构。在这种存储结构中,矩阵中的每个非零元素用一个结点来表示;这种非零元素结点有五个域组成,其结构如图 6-10(a)所示。

其中行域(row)、列域(right)、值域(val)分别表示一个非零元素所在的行号、列号和值;向下域(down)用以链接同一列中下一个非零元素的结点;向右域(right)用以链接同一行中下一个非零元素的结点。这样,同一行上的非零元素结点可通过 right 域链接成一个带表头结点的循环链表,称为行循环链表;同一列上的非零元素结点也可通



(a) 非零元素结点　(b) 行(列)表头结点　(c) 总表头结点

图 6-10　十字链表中的三种结点结构

过 down 域链接成一个带表头结点的循环链表，称为列循环链表。为了使整个链表中的结点结构一致，我们规定行(列)循环链表中的表头结点和表示非零元素结点一样，也设五个域，如图 6-10(b)所示。其中，相应的行域(row)和列域(col)规定为零；next 域用以链接下一行(列)循环链表的表头结点；down 域用以链接所在列循环链表中第一个非零元素结点；right 域用以链接所在行循环链表中第一个非零元素结点。从中可以看出每一列的列循环链表中表头结点只需用一个 down 域，每一行的行循环链表中只需用一个 right 域。由于这些表头结点的 row 域、col 域的值均为零，所以这两组表头结点可以合用(即第 i 行的链表和第 i 列的链表共用一个表头结点)。而这些行(列)表头结点又可通过 next 域链接成一个带头结点的循环链表，称为表头结点循环链表；这个循环链表的头结点称为总头结点，它也设五个域，如图 6-10(c)所示。其中，row 域和 col 域分别用于存放矩阵的行数和列数；next 域用于链接第一行(列)链表的表头结点；另外两个域(down、right)在总表头结点中不用。

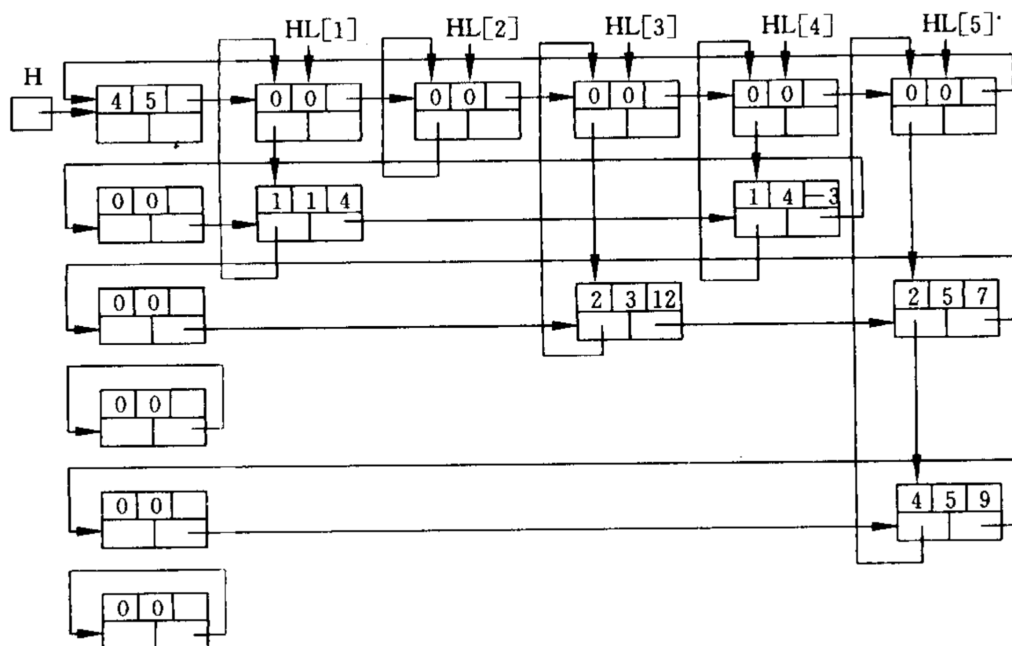
总之，在这种多重链表中有三种结点：即总表头结点、行(列)表头结点和非零元素结点；另外还有三种循环链表：由各行(列)表头结点组成的表头结点循环链表、由同一行中的非零元素结点组成的行循环链表和由同一列中的非零元素结点组成的列循环链表。

在这种稀疏矩阵的多重链表存储结构中，对于每一个非零元素 aij 的结点，它既是第 i 行循环链表中的一个结点，又是第 j 列循环链表中的一个结点，就好比处在一个十字路口上，所以称这种多重链表为十字链表。如图 6-11(a)所示的稀疏矩阵 M，其十字链表存储结构如图 6-11(b)所示。

在图 6-11 中，h 是指向总表头结点的指针。通过 h 我们便可以

$$M=\begin{bmatrix} 4 & 0 & 0 & -3 & 0 \\ 0 & 0 & 12 & 0 & 7 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 9 \end{bmatrix}$$

(a) 稀疏矩阵M



(b) 矩阵M的十字链表

图 6-11　稀疏矩阵的十字链表存储结构

得到该链表中所有非零元素的信息。现在我们讨论如何建立一个稀疏矩阵的十字链表问题。在建立一个稀疏矩阵的十字链表时，首先要知道矩阵的行数、列数和非零元素数目；并且要知道每个非零元素的行号、列号和值。下面我们先大致描述一下这个算法的思想。(假设输入数据都是正确的)。

1. 首先输入矩阵的行数 m、列数 n 和非零元素数目 t；然后取 m 和 n 两者中较大一个赋给 s(s=max{m,n})；最后再建立一个总表头结点和 s 个行(列)链表的表头结点。为了能方便地随机访问各行(列)表头结点，另外设立一个向量 hl[1..s]，其每个分量 hl[i]指向第 i 个行(列)链表的表头结点。

2. 组织循环，以行优先序依次输入矩阵的非零元素三元组(r,c,v)。每输入一个三元组，先建立一个结点 p，再把这个结点插入到行循环链表和列循环链表中。由于规定以行优先序输入非零元素，因

而结点 p 一定插在第 r 行循环链表和第 c 列循环链表的当前最后结点之后。为了能快速地找到插入位置，我们暂时借用每个行(列)链表的表头结点中的 next 域，用于指示每一列循环链表中当前最后一个结点；并用一个变量 rl 记录最近处理的行号，用一个指针 ep 指示最近处理的行循环链表中的最后一个结点。如果当前输入的非零元素的行号(r)大于最近处理的行号(rl)，则修改 rl 和 ep。这样，在行循环链表中插入时只要把结点 p 插在 ep 所指结点之后；在列循环链表中插入时只要把结点 p 插在 hl[c]↑.next 所指结点之后。

3. 完成表头结点循环链表的链接。此算法的框图描述如图6-12所示：

根据此框图，下面我们用 PASCAL 语言描述有关数据类型和此算法。

```
CONST nmax={允许的最大行数和列数}；
TYPE link=↑node；
     feature=(headnode,element)；
     node=RECORD
              row,col:integer；
              right,down:link；
              CASE feature OF
                headnode:(next:link)；
                element:(val:integer)
              END
            END；
PROCEDURE crt-linkmat(VAR h:link)；
VAR
  m,n,s:1..nmax；
  hl:ARRAY [1..nmax] OF link；
  t,r,c,v,rl,i:integer；
  p,ep:link；
BEGIN
  readln(m,n,t)；
  IF m>n THEN s:=m
```

```
            ELSE s:=n;
    new(h);
    h↑.row:=m; h↑.col:=n;
    FOR i:=1 TO s DO
      BEGIN
        new(hl[i]);
        hl[i]↑.row:=0; hl[i]↑.col:=0;
        hl[i]↑.right=hl[i]; hl[i]↑.down:=hl[i];
        hl[i]↑.next:=hl[i]
      END;
    rl:=1 ; ep:=hl[1];
    FOR i:=1 TO t DO
      BEGIN
        readln(r,c,v);
        new(p);
        p↑.row:=r; p↑.col:=c; p↑.vol:=v;
        IF rl<r THEN
          BEGIN
            rl:=r; ep:=hl[r]
          END;
        p↑.right:=ep↑.right;
        ep↑.right:=p; ep:=p;
        p↑.down:=hl[c]↑.next↑.down;
        hl[c]↑.next↑.down:=p;
        hl[c]↑.next:=p
      END;
    FOR i:=1 TO s-1 DO hl[i]↑.next:=hl[i+1];
    h↑.next:=hl[1];
    hl[s]↑.next:=h
  END;
```

在这个算法中，共有 3 个并列的单循环，所以其时间复杂度为 O(s+t)。下面我们讨论用十字链表作为稀疏矩阵的存储结构时，如何实现两个矩阵的相加运算 A:=A+B。

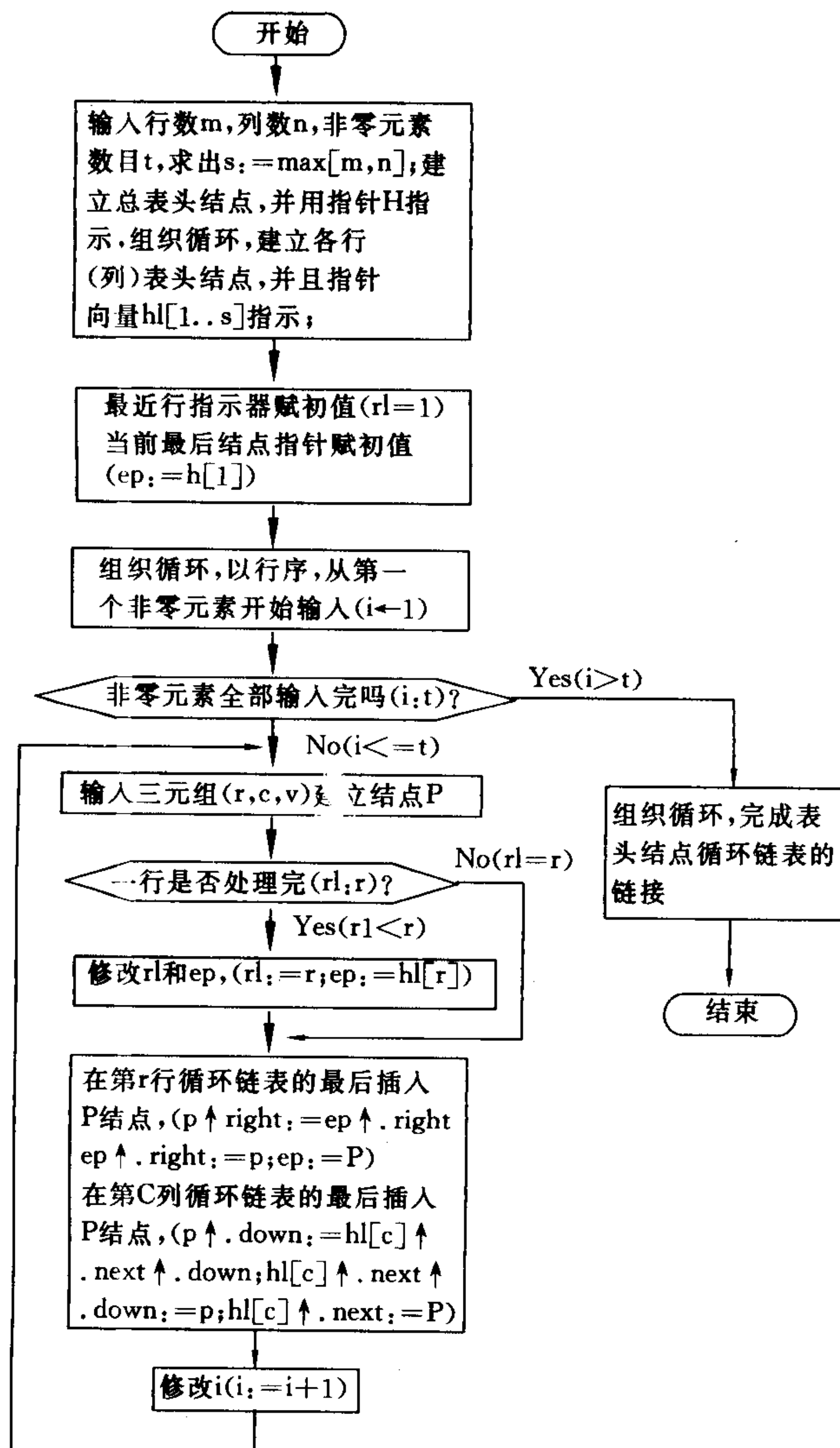


图 6-12 建立十字链表的算法框图

两个矩阵相加和第三日中讨论的两个一元多项式相减极为相似,所不同的是一元多项式中的每个项只有一个标元(即指数),而矩

阵中的每个非零元有两个标元(行值与列值),每个结点既在行表中又在列表中,致使插入和删除时指针的修改稍为复杂,故需更多的辅助指针。

假设矩阵 $A':=A+B$,则结果矩阵 A' 中的非零元 a_{ij}' 只可能有三种情况。它或者是 $a_{ij}+b_{ij}$;或者是 a_{ij}($b_{ij}=0$ 时);或者是 b_{ij}($a_{ij}=0$ 时)。由此,当将 B 加到 A 上去时,对 A 矩阵的十字链表来说,或者是改变结点的 val 域值($a_{ij}+b_{ij}<>0$),或者不变($b_{ij}=0$),或者插入一个新结点($a_{ij}=0$ 且 $b_{ij}<>0$),还可能是删除一个结点($a_{ij}+b_{ij}=0$)。整个运算过程可从矩阵的第一行起逐行进行。对每一行都从行表头出发分别找到 A 和 B 在该行中第一个非零元结点后开始比较,然后按上述四种不同情况分别处理之(假设指针 pa 和 pb 分别指向 A 和 B 的十字链表中行值相同的两个结点)。

(1) pa↑.col=pb↑.col 且 pa↑.val+pb↑.val<>0,则只要将 $a_{ij}+b_{ij}$ 的值送到 pa 所指结点的值域中即可,其它所有域的值都不变。

(2) pa↑.col=pb↑.col 且 pa↑.val+pb↑.val=0,则需要在 A 矩阵的链表中删除 pa 所指的结点。此时,需改变同一行中前一结点的 right 域值,以及同一列中前一结点的 down 域值。

(3) pa↑.col<pb↑.col 且 pa↑.col<>0,则只要将 pa 指针往右推进一步,并重新加以比较。

(4)pa↑.col>pb↑.col 或 pa↑.col=0,则需在 A 矩阵的链表中插入一个值为 b_{ij} 的结点。此时需改变相应的指针。

为了便于插入和删除结点,还需设立一些辅助指针。其一,在 A 的行链表上设 qa 以指示 pa 所指结点的前驱结点;其二,在 A 的每一列的列链表上设一个指针 hl[j]它的初值是指向每一列的列链表的表头结点。

下面对将矩阵 B 加到矩阵 A 上的操作过程大致描述如下:

设 ha 和 hb 分别为表示矩阵 A 和 B 的十字链表的头指针;ca 和 cb 分别为指向 A 和 B 的行链表的表头结点的指针,其初始状态为:

ca:=ha↑.next;cb:=hb↑.next;

pa 和 pb 分别为指向 A 和 B 的链表中结点的指针。

（1）令 pa 和 pb 分别指向 A 和 B 的链表中第一行的第一个非零元素的结点，即

pa:=ca↑.right;pb:=cb↑.right;

若 B 的该行中无非零元素的结点，即 pb↑.col=0，则令 ca 和 cb 各自转指向下一行，即

ca:=ca↑.next;cb:=cb↑.next;

（2）否则，比较这两个结点的列序号，这时可能有三种情况：

- 若 pa↑.col<pb↑.col 且 pa↑.col<>0，则令 pa 指向本行下一结点，即

  qa:=pa;pa:=pa↑.right;

- 若 pa↑.col>pb↑.col 或 pa↑.col=0（即 A 的这一行中非零元已处理完），则需在 A 中插入一个结点。假设新结点由指针 p 指示，则 A 的行表中的指针变化状况为：

  qa↑.right:=p;p↑.right:=pa;

  A 的列表中的指针也要作相应的改变。首先需找到同一列中上一个结点，并且令 hl[j]指向该结点。于是 A 的列表指针修改为：

  p↑.down:=hl[j]↑.down;hl[j]↑.down:=p;

- 若 pa↑.col=pb↑.col，则将 B 中的这个非零元的值加到 A 的相应非零元上，即

  pa↑.val:=pa↑.val+pb↑.val;

  此时若 pa↑.val<>0，则指针不变；否则，需删除 A 中该结点。于是行表中指针变为：

  qa↑.right=pa↑.right;

  同时，为了改变列表中的指针，需要先找同一列中上一个结点，且令 hl[j] 指向该结点，然后令：

  hl[j]↑.down:=pa↑.down;

（3）重复步骤（2），直至 B 的同一行中非零元处理完为止，然后再转向下一行。以此类推，直至 m 行都处理完为止。此时的标志是 ca 和 cb 重又指向十字链表的头结点，即

$$ca=ha \quad 或 \quad cb=hb;$$

该算法的框图描述如图 6-13 所示：

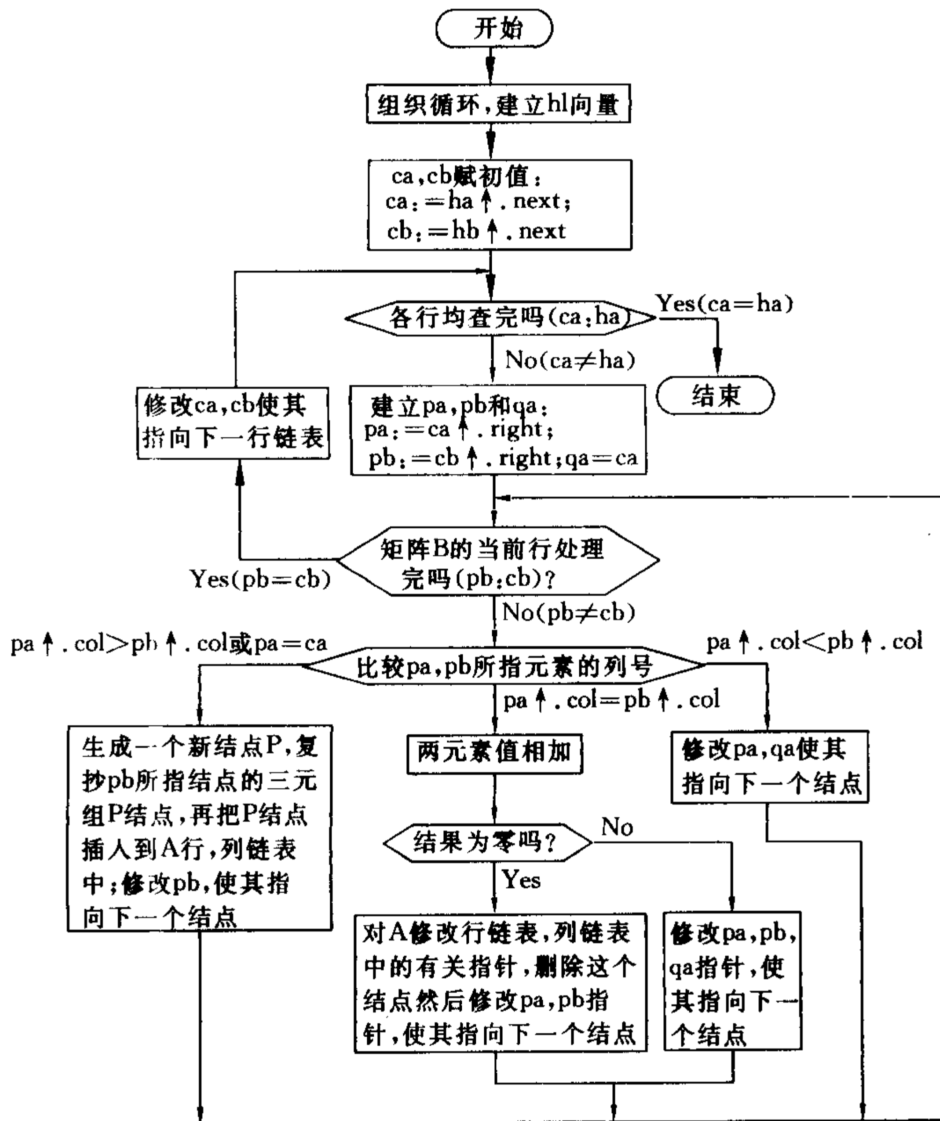


图 6-13　十字链表表示的稀疏矩阵相加算法框图

根据此框图，下面我们用 PASCAL 语言描述此算法。

```
PROCEDURE add-linkmat(VAR ha:link; hb:link);
VAR
  pa,pb,qa,ca,cb,q:link;hl:ARRARY[1...nmax] OF link;
  i,j:integer;
BEGIN
```

```
p:=ha↑.next; i:=1;
WHILE p<>ha DO
  BEGIN
    hl[i]:=p;
    p:=p↑.next;
    i:=i+1
  END;
ca:=ha↑.next; cb:=hb↑.next;
WHILE ca<>ha DO
  BEGIN
    pa:=ca↑.right; pb:=cb↑.right; qa:=ca;
    WHILE pb<>cb DO
      IF (pa↑.col>pb↑.col) OR (pa=ca) THEN
        BEGIN
         new(p); p↑.row:=pb↑.row; p↑.col:=pb↑.col;
         p↑.val:=pb↑.val;
         qa↑.right:=p; p↑.right:=pa; qa:=p;
         j:=p↑.col; q:=hl[j]↑.down;
         WHILE (q↑.row<>0) AND (q↑.row<p↑.row) DO
          BEGIN
            hl[j]:=q; q:=q↑.down
          END;
         hl[j]↑.down:=p; p↑.down:=q;
         hl[j]:=p;
         pb:=pb↑.right
        END
      ELSE
        IF (pa↑.col<pb↑.col) THEN
          BEGIN
            qa:=pa; pa:=pa↑.right
          END
        ELSE
          BEGIN
```

```
              pa↑.val:=pa↑.val+pb↑.val
              IF pa↑.val=0 THEN
                BEGIN
                  qa↑.right:=pa↑.right;
                  j:=pa↑.col; q:=hl[j]↑.down;
                  WHILE (q↑.row<pa↑.row) DO
                    BEGIN
                      hl[j]:=q; q:=q↑.down
                    END;
                  hl[j]↑.down:=q↑.down;
                  dispose(pa);
                  pa:=qa↑.right; pb:=pb↑.right
                END
              ELSE
                BEGIN
                  qa:=pa; pa:=pa↑.right;
                  pb:=pb↑.right
                END
            END;
      ca:=ca↑.next; cb:=cb↑.next
    END
  END;
```

从上述过程可以看出，对于一个结点来说，进行比较、修改指针所需的时间是一个常数。整个过程主要是对 ha 和 hb 两个十字链表的逐行扫描，因此，其循环次数主要取决于 A 和 B 矩阵中非零元素的个数 ta 和 tb；因此算法的时间复杂度为 O(ta+tb)。

习　　题

1. 编写一个用三元组形式表示的两个稀疏矩阵 M 和 N 相乘的算法。结果存放在二维数组中。
2. 写一个算法，其功能为访问稀疏矩阵的十字链表中每一个元

素，访问时要求打印元素的行号、列号和元素值。

3. 若在 $m \times n$ 的矩阵中某一个元素 a_{ij} 满足以下条件：a_{ij} 既是第 i 行各元素中的最小值，又是第 j 列各元素中的最大值，则称此为矩阵的鞍点。试写一个算法，求出矩阵中的鞍点；若该矩阵不存在鞍点，则给出相应的信息。

4. 将下列稀疏矩阵 A 表示成十字链表。

$$A=\begin{bmatrix}0 & 1 & 0 & 0 & 5\\0 & 0 & 0 & 4 & 0\\0 & 0 & 0 & 0 & 0\\2 & 0 & 0 & 0 & 0\\0 & 3 & 0 & -2 & 0\end{bmatrix}$$

第七日　树

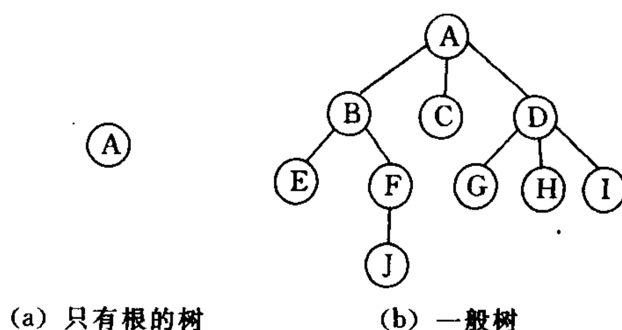
前面，我们主要讨论了线性表、栈、队列等一些线性结构。然而，在实际应用中我们还常常会碰到一些问题，需要用非线性结构描述。所谓非线性结构就是说在这种结构中，至少存在一个元素有两个或两个以上的直接前驱(或后继)。树型结构就是其中一类重要的非线性数据结构。本日主要讨论二叉树的存储结构及其各种操作，并研究树和森林与二叉树之间的转换关系，最后将介绍几个有关二叉树的典型应用例子。

7.1　树型结构的基本概念和基本操作

一、树的定义和表示形式

树是 $n(n \geqslant 0)$个结点的有限集。当 $n=0$ 时，称为空树；当 $n>0$ 时，一棵树中，有且仅有一个特定的称为根的结点，而除根以外的其他结点可分为 $m(m \geqslant 0)$个互不相交的有限集 T_1、T_2、…、T_m，其中每个有限集本身又是一棵树，并且称为根的子树。例如，在图 7-1 中，(a)是只有一个根结点的树；(b)是有 10 个结点的树，其中 A 是根，其余结点分成三个互不相交的有限集：$T_1=\{B,E,F,J\}$，$T_2=\{C\}$，$T_3=\{D,G,H,I\}$。T_1，T_2，T_3 本身又都是一棵树，都称为根结点 A 的子树。例如，T_1 是一棵树，其根为 B，其余结点分成两个互不相交的有限集 $T_{11}=\{E\}$，$T_{12}=\{F,J\}$。T_{11}，T_{12}都是 B 的子树，T_{11}是只有根结点 E 的树；在树 T_{12}中，F 为根，F 有一棵只有根结点 J 的子树。

在树的定义中又用到了树的概念，所以这是一个递归的定义；这



(a) 只有根的树　　　　(b) 一般树

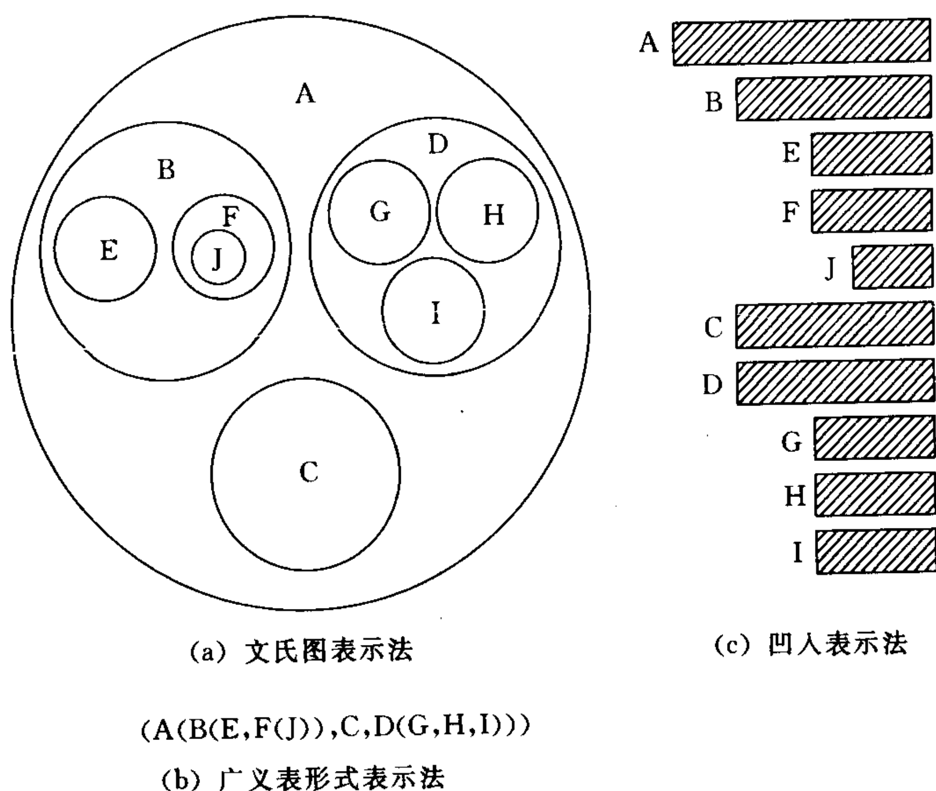
图 7-1　树的示例

也说明了树的固有特性，在树中每一个结点都是该树中的某一棵子树的根。

对于一棵树，我们有多种表示形式。图 7-1 所示是一种树形表示法，它将根结点画在最上面，形似一棵倒悬的自然界中的树。在本书中一般都采用这种表示法。除此以外，还有其他一些表示形式。图7-2所示为图 7-1 中(b)树的其他表示形式。其中，(a)为嵌套集合的文氏图表示法，(b)为广义表形式表示法，(c)为凹入表示法。

二、树的基本术语

树的结点包含一个数据元素以及若干指向其子树的分支。结点拥有的子树数称为结点的度。例如，在图 7-1(b)中结点 A 的度为 3，结点 B 的度为 2，结点 C 的度为 0。我们把度为 0 的结点称为叶子(或终端结点)。在图 7-2(b)中，结点 C、G、H、I、E、J 都是叶子。另外我们把树中度不为 0 的结点称为非终端结点(或分支结点)。除根结点之外的分支结点称为内结点。在树中各结点的最大值称为树的度。如图 7-1(b)所示的树，其度为 3。树中任一结点的子树的根称为该结点的孩子；反之，该结点称为其孩子的双亲。例如，在图 7-1(b)中，结点 B 为结点 A 的子树 T_1 的根，所以 B 称为 A 的孩子，A 称为 B 的双亲。同一双亲的孩子之间互称为兄弟。例如，G、H、I 互为兄弟。将这些关系进一步推广，可认为 B 是 J 的祖先。结点的祖先是从根到该结点所经分支上的所有结点。例如，J 的祖先为 A、B、F。反之以某结点为根的子树中的所有结点都称为该结点的子孙。例如，B 的子孙为



(a) 文氏图表示法

(c) 凹入表示法

(b) 广义表形式表示法

图 7-2　树的其它三种表示形式

E、F、J。

结点的层次从根开始定义,根为第一层,根的孩子为第二层;若某结点在第 k 层,则其孩子就在第 k+1 层。在一棵树中,双亲在同一层的结点互称为堂兄弟。例如,结点 E、F、G、H、I 互为堂兄弟。树中结点的最大层次称为树的深度(或高度)。图 7-1(b)所示的树,其深度为 4。

在树 T 中,如果各结点的子树间的相对次序是有意义的,则称 T 为有序树;否则,称为无序树。对于有序树,改变子树间的相对次序,就变成了另外一棵树。

森林是 m(m≥0)棵互不相交的树的集合。森林的概念与树非常接近,只要把一棵树的根结点去掉,就可以变成森林。例如,如图 7-1(b)中的树,去掉 A 就成为由三棵树组成的森林;反之,如果把由 m 棵树组成的森林,加上一个根结点而把这 m 棵树作为此根的子树,则使森林变成了树。

三、树的基本操作

树的基本操作有下列几种：

1. 初始化(initiate(T))：置 T 为空树。
2. 求根函数(root(T)或 root(x))：求树 T 的根或求结点 x 所在的树的根结点。若 T 是一棵空树或 x 不在任何一棵树上，则函数值为“空”。
3. 求双亲函数(parent(T,x))：求树 T 中结点 x 的双亲结点。若结点 x 是树 T 的根结点或结点 x 不在树 T 中，则函数值为“空”。
4. 求孩子结点(child(T,x,i))：求树 T 中结点 x 的第 i 个孩子结点。若结点 x 是树 T 的叶子或无第 i 个孩子或结点 x 不在树 T 中，则函数值为“空”。
5. 求右兄弟函数(right-sibling(T,x))：求树 T 中结点 x 右边的兄弟。若结点 x 是其双亲的最右边的孩子结点或结点 x 不在树 T 中，则函数值为“空”。
6. 建树(crt-tree(x,F))：生成一棵以 x 结点为根，以森林 F 为子树森林的树。
7. 插入子树操作(ins-child(y,i,x))：置以结点 x 为根的树为结点 y 的第 i 棵子树 。若原树中无结点 y 或结点 y 的子树个数 $<$i-1，则空操作。
8. 删除子树操作(del-child(x,i))：删除结点 x 的第 i 棵子树。若无结点 x 或结点 x 的子树个数少于 i，则空操作。
9. 遍历操作(traverse(T))：按某个次序依次访问树中各个结点，并使每个结点只被访问一次。

树的应用十分广泛，在不同的应用中树的操作不尽相同。因此，树的存储结构可随需要而设定。

四、树的存储结构

树的存储结构根据应用的不同，有多种形式。在此，我们介绍如

下三种比较常用的方法。

1. 双亲表示法

在这种方法中，用一组连续的存储单元存储树中的结点，结点的形式如下：

data	parent

其中，data 用于存放有关结点本身的信息，parent 用于指示该结点的双亲位置。例如，图 7-1(b)所示的树，其双亲表示法如图 7-3 所示。

	data	parent
1	A	0
2	B	1
3	C	1
4	D	1
5	E	2
6	F	2
7	G	4
8	H	4
9	I	4
10	J	6

data	parent

结点形式

图 7-3 树的双亲表示法

这种存储结构利用了每个结点(除根以外)只有唯一双亲的性质。在这种存储结构下，求结点的双亲十分方便，也很容易求树的根。但是在这种表示法中，在求结点的孩子时需要遍历整个向量。

2. 孩子表示法

由于树中每个结点可能有多个孩子，所以自然想到用多重链表存储树。在这种多重链表的每个结点中除了用于存放数据信息的 data 域外，还有若干指针域，分别用于指向该结点的孩子结点。但是一棵树中，不同结点的度数是不同的，那么每个结点中到底需要多少个指针域呢？我们有如下二种方案：

(1) 定长结点的多重链表

在这种方法中，我们取树的度数作为每个结点的指针域数目。不难推导，一棵具有 n 个结点的度为 k 的树中必有 $n(k-1)+1$ 个空指针。显然，这会造成存储空间的极大浪费。例如，图 7-1(b)所示的树，其定长结点的多重链表如图 7-4(a)所示。

(2) 不定长结点的多重链表

在定长结点的多重链表中，由于各结点指针的数目是根据树的度而定，所以造成了存储空间的浪费。下面我们考虑每个结点的指针域数目取它自已的度数；另外，在每个结点中设置一个度数域(de-

gree)，以指出该结点的度数，其表示方法如图 7-4(b)所示。显然这种方法的存储密度较前者有所提高；但由于各结点的结构不同，自然会造成操作上的不方便。

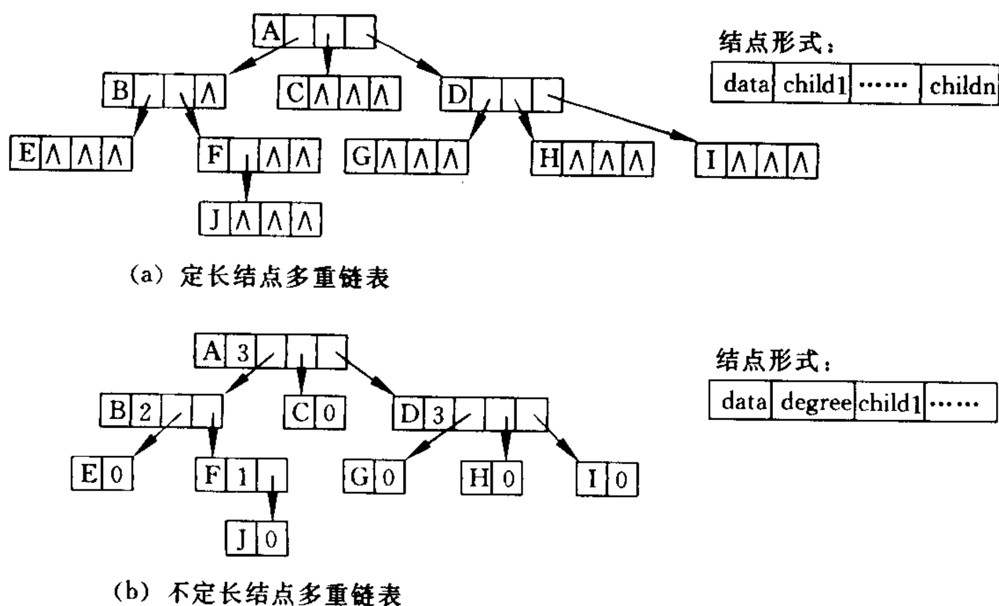


图 7-4 树的多重链表存储结构

3. 孩子-兄弟表示法

这种方法又称二叉树表示法(或二叉链表表示法)。在这种二叉链表的每个结点中除了用于存放数据信息的 data 域外，还有两个指针域 fch 和 nsib，分别用于指向该结点的第一个孩子结点和它的下一个兄弟结点。图 7-5 为图 7-1(b)所示树的孩子-兄弟链表。在这种存储结构中，树的操作比较方便，且存储密度较高。

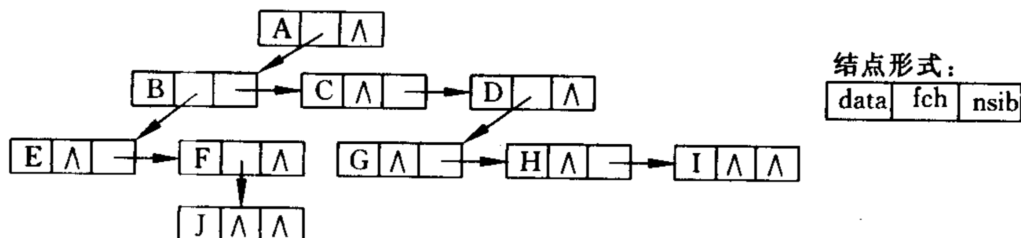


图 7-5 树的孩子-兄弟链表存储结构

7.2 二 叉 树

二叉树是一种应用广泛的特殊的树形结构，它的特点是每个结点最多只能有两个孩子。在二叉树中，必须严格区分左右孩子，其次序不能颠倒，若改变次序则变成另一棵二叉树。

一、二叉树的定义和基本操作

二叉树是 n(n≥0)个结点的有限集合。当 n=0 时称为空二叉树；否则，二叉树是由一个根结点和至多两棵互不相交的被称为该根的也为二叉树的左子树和右子树组成。图 7-6 列出了二叉树的五种基本形态。

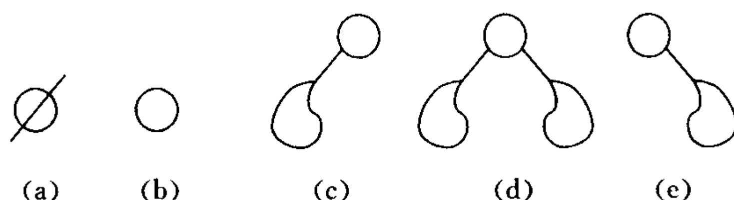


图 7-6 二叉树的五种基本形态

其中，(a)为空二叉树；(b)为只有根结点的二叉树；(c)为右子树为空的二叉树；(d)为左右子树都非空的二叉树；(e)为左子树为空的二叉树。

二叉树的基本操作和树的基本操作类似。下面我们分别介绍：

1. 初始化操作(initiate(BT))：置 BT 为空二叉树。
2. 求根函数(root(x))：求结点 x 所在二叉树的根结点。若 x 不在任何二叉树上，则函数值为"空"。
3. 求双亲函数(parent(BT,x))：求二叉树 BT 中结点 x 的双亲结点。若结点 x 是二叉树 BT 的根结点或二叉树 BT 中无 x 结点，则函数值为"空"。
4. 求孩子结点函数(lchild(BT,x)及 rchild(BT,x))：分别求二叉树 BT 中结点 x 的左孩子及右孩子结点。若结点 x 没有左孩子及右孩子或 x 不在二叉树 BT 中，则函数值为"空"。

5. 求兄弟函数(lsibling(BT,x)及 rsibling(BT,x)):分别求二叉树 BT 中结点 x 的左兄弟及右兄弟结点。若结点 x 是根结点或不在 BT 中、或其没有左兄弟及右兄弟,则函数值为“空”。
6. 建树操作(crt-bt(x,LBT,RBT)):生成一棵以结点 x 为根,二叉树 LBT、RBT 分别为其左、右子树的二叉树。
7. 插入子树操作(ins-lchild(BT,y,x)及 ins-rchild(BT,y,x)):将以结点 x 为根且右子树为空的二叉树分别置为二叉树 BT 中结点 y 的左子树及右子树。若结点 y 已经有左子树及右子树,则插入后,原来 y 的左子树及右子树置为 x 的右子树;若 y 不在 BT 中,则为空操作。
8. 删除子树操作(del-lchild(BT,x)及 del-rchild(BT,x)):分别删除二叉树 BT 中以结点 x 为根的 x 的左子树及右子树。若 x 无左子树及右子树、或 x 不在 BT 中,则为空操作。
9. 遍历操作(traverse(BT)):按某个次序依次访问二叉树中各个结点,并使每个结点只被访问一次。

二、二叉树的性质

二叉树具有许多重要性质,下面作简单的讨论:

1. 在一棵二叉树中,第 i 层的结点数最多为 2^{i-1}($i \geq 1$)。

例如	层次 i	第 i 层最多结点数
	1	$2^{1-1}=1$
	2	$2^{2-1}=2$
	3	$2^{3-1}=4$
	.	.
	.	.
	.	.
	h	2^{h-1}

这个性质可以用数学归纳法进行证明:

当 $i=1$ 时,只有一个根结点,显然 $2^{i-1}=2^0=1$,命题成立。

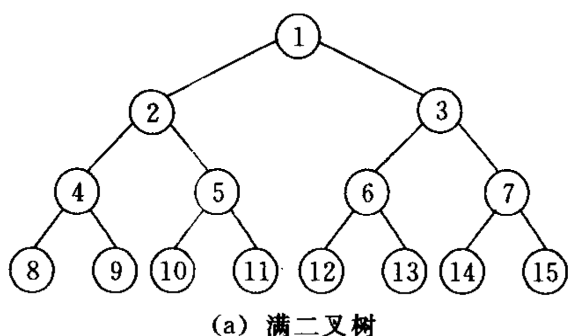
现假设对所有的 i,$1<=i<k$,命题成立,即第 $k-1$ 层上最多有

2^{k-2}个结点。由于二叉树中每个结点的度最多为 2,所以第 k 层上的最大结点数为第 k－1 层上的最大结点数的 2 倍,即 $2*2^{k-2}=2^{k-1}$。由此可见,命题成立。

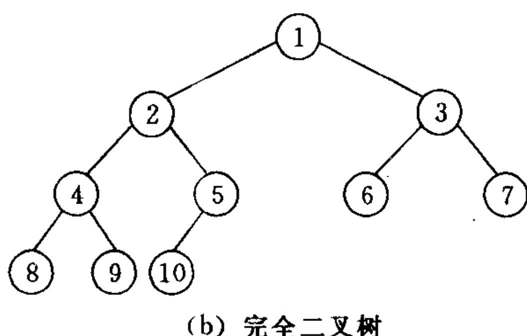
2. 深度为 h 的二叉树上结点总数最多为 2^h-1

由性质 1 可见,若将二叉树中每一层上结点的最大数相加,结果就为二叉树上结点的最大数。

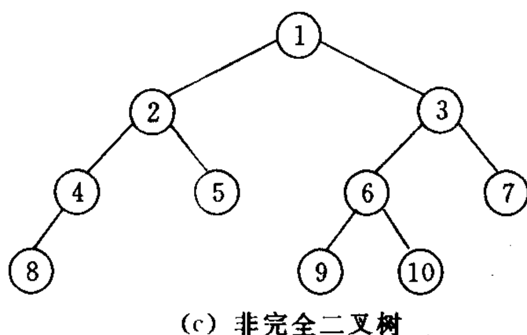
$$\sum_{i=1}^{h}(\text{第 i 层上的最大结点数})=\sum_{i=1}^{h}2^{i-1}=2^h-1$$



(a) 满二叉树



(b) 完全二叉树



(c) 非完全二叉树

图 7-7　二叉树示例

如果一棵深度为 h 的二叉树,共有 2^h-1 个结点,则此二叉树称为满二叉树。若对一棵满二叉树,从第 1 层开始,自上而下,从左到右地对结点进行连续编号,就得到了二叉树结点的顺序编号。例如,图 7-7(a)是一棵深度为 4 的满二叉树,并对结点进行了顺序编号。

如果深度为 h 的有 n 个结点的二叉树,能够与深度为 h 的顺序编号的满二叉树从编号 1 到 n 的结点相对应,则这样的二叉树称为完全二叉树。图 7-7(b)所示为一棵深度为 4 的完全二叉树。可以看出,在完全二叉树中,叶子结点只可能在层次最大的两层上出现;并且对于树中的任一结点,若其右分支下子孙的最大层次为 l,

则其左分支下子孙的最大层次为 l 或 l+1。但满足这二点的二叉树不一定是完全二叉树，如图 7-7(c)所示为一棵非完全二叉树。

3. 对于任何一棵二叉树 BT，设 n_0、n_1、n_2 分别是其度数为 0、1、2 的结点数。则有：$n_0=n_2+1$

证明：因为在二叉树中，任何结点的度均小于或等于 2，所以二叉树中结点总数为：

$$n=n_0+n_1+n_2 \tag{7-1}$$

设 B 为二叉树中分支总数目。由于在二叉树中除根结点外，其余结点都有一个分支进入，所以，$B=n-1$，即：$n=B+1$。而这些分支只能是由度为 1 或 2 的结点所发出，所以，$B=n_1+2n_2$，即：

$$n=n_1+2n_2+1 \tag{7-2}$$

由(8－1)式和(8－2)式得：$n_0=n_2+1$

4. 具有 n 个结点的完全二叉树的深度为$\lfloor \log2n \rfloor+1$

设其深度为 h，根据性质 2 和完全二叉树的定义有：

$2^{h-1}-1<n<=2^h-1$　　　　即 $2^{h-1}<=n<2^h$，所以 $h-1<=\log_2 n<h$，又因为 h 是整数，所以 $h=\lfloor \log2n \rfloor+1$。

5. 如果对一棵具有 n 个结点的完全二叉树的结点进行顺序编号(自上而下，从左到右进行编号)，则其中任一结点 $i(1\leqslant i\leqslant n)$有：

(1) 如果 $i=1$，则结点 i 是二叉树的根，无双亲；如果 $i>1$，则其双亲 parent(i)的编号为$\lfloor i/2 \rfloor$。

(2) 如果 $2i>n$，则结点 i 无左孩子(结点 i 为叶子结点)；否则其左孩子 lchild(i)的编号为 2i。

(3) 如果 $2i+1>n$，则结点 i 无右孩子；否则右孩子 rchild(i)的编号为 $2i+1$。下面先用数学归纳法证明(2)和(3)：

对于 $i=1$，由完全二叉树的定义，其左孩子编号为 2，右孩子编号为 3。如果 $2>n$，则说明 i 无左、右孩子；如果 $3>n$，则说明 i 无右孩子，可见命题(2)和(3)成立。

设 $i=k(k>=1)$时，命题(2)和(3)成立。即 $2k+1<n$ 时，k 的左孩子编号为 2k、右孩子编号为 $2k+1$。则当 $i=k+1$ 时，若它有左孩子，则编号必为$(2k+1)+1=2(k+1)$；若它有右孩子，则编号必为

$(2k+1)+2=2(k+1)+1$，所以命题(2)和(3)成立。

由命题(2)和(3)，显然可推出命题(1)。

三、二叉树的存储结构

对于二叉树我们可以用顺序存储结构存储，也可以用链式存储结构存储。下面分别进行讨论。

1. 二叉树的顺序存储结构

对二叉树进行顺序存储时，需要用一组连续的存储单元存储二叉树的结点中的数据。例如，图 7-7(b)所示的完全二叉树，可以用向量 bt[1..10]作它的存储结构，将二叉树中编号为 i 的结点中的数据存放在分量 bt[i]中，如图 7-8(a)所示。

1	2	3	4	5	6	7	8	9	10

(a) 完全二叉树

1	2	3	4	5	6	7	8	0	0	0	9	10

(b) 非完全二叉树

图 7-8　二叉树的顺序存储结构

根据完全二叉树的特性，结点在向量中的相对位置蕴含着结点之间的关系。如 bt[4]的双亲存放在 bt[2]中，而其左右孩子存放在 bt[8]和 bt[9]。对于满二叉树和完全二叉树，用这种存储结构显然是比较适合的。它既不浪费内存，又可以利用地址公式确定其结点的存储位置。然而，对于一般的二叉树而言，顺序存储常常会造成内存的浪费。例如，图 7-7(c)所示的二叉树，其顺序存储结构如图 7-8(b)所示。其中有三个单元浪费。在最坏的情况下，一棵深度为 h 只有 h 个结点的单支树（树中无度为 2 的结点），需要 2^h-1 个存储分量。如图 7-9(a)所示的二叉树，其顺序存储结构如图 7-9(b)所示。该二叉树只有四个结点，却需开辟 15 个结点的空间。显然，空间浪费很大。

另外，对于二叉树顺序存储结构，插入和删除是很不方便的。所以一般情况下，二叉树采用链式存储结构。

2. 二叉树的链式存储结构

在二叉树的链式存储结构中，二叉树中的每一个结点在链表中用一个结点表示。每个结点有多少个域可以根据需要来设计，一般定



(a) 单支树　　　　(b) 单支树的顺序存储结构

图 7-9　单支树及其顺序存储结构

为三个域,如图 7-10(a)所示。其中,data 为数据域,用于存储二叉树的结点中的数据;lchild、rchild 分别为左、右指针域,用于指向表示其左、右孩子的结点。有时,为了便于寻找结点的双亲,还可以在结点结构中增加一个指向其双亲结点的指针域:parent,其结点结构如图 7-10(b)所示。采用这两种结点结构所构成的二叉树存储结构分别称之为二叉链表和三叉链表。例如,图 7-10(c)所示的二叉树,其二叉链表、三叉链表存储结构分别如图 7-10(d)和图 7-10(e)所示。

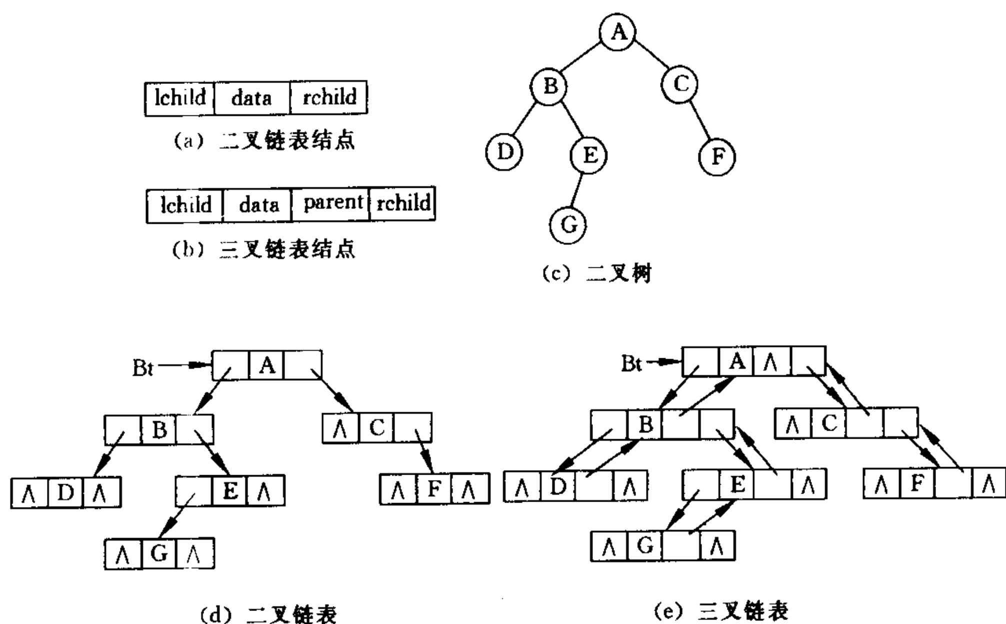


图 7-10　二叉树链式存储结构

在二叉树的链式存储结构中,我们需用一个头指针指向根结点。通过头指针,可以获得其它任一结点的信息。显然,在二叉树的二叉链表存储结构中,存在一些空指针域。因为含有 n 个结点的二叉链

表，总共有 2n 个指针域（一个结点有 2 个指针域）；而 n 个结点只需要 n－1 个分支相连接，一个分支对应一个指针，故仅需 n－1 个指针，所以 n 个结点的二叉链表中有 n+1 个空指针域。在后面我们将介绍可利用这些空指针域来存储其它一些信息，从而可获得二叉树的另一种链式存储结构。

7.3 遍历二叉树

遍历二叉树是指以一定的次序，访问二叉树中的每个结点，且每个结点只被访问一次。在这里“访问”的含义很广，可以理解为输出结点中数据的值或对结点中的数据进行处理等。遍历二叉树的过程实际上就是把二叉树中的结点进行线性排列。由于二叉树中有两种分支，所以遍历的次序不同得到的结果也就不同。设 L、D、R 分别表示遍历左子树、访问根结点、遍历右子树，则对一棵二叉树有六种遍历方案：DLR、LDR、LRD、DRL、RDL、RLD。如果在左、右子树的遍历次序上规定先左后右，则只有前三种遍历方案，分别称之为：先根（序）遍历（DLR）、中根（序）遍历（LDR）、后根（序）遍历（LRD）。

在讨论二叉树遍历的实现之前，先给出二叉链表结点的 PASCAL 语言类型描述：

```
TYPE bitreptr= ↑ bnode;
     bnode=RECORD
              data:char;
              lchild,rchild:bitreptr
             END;
```

下面介绍在二叉链表上实现二叉树遍历的算法。

一、先根遍历

先根遍历又称为先序遍历，其递归定义是：若二叉树为空，则空操作；否则，依次执行以下操作：

1. 访问根结点；

2. 先根遍历左子树；

3. 先根遍历右子树。

此递归过程算法的 PASCAL 语言描述如下：

```
PROCEDURE preorder(bt:bitreptr);
  BEGIN
    IF bt<>nil THEN
      BEGIN
        write(bt↑.data);
        preorder(bt↑.lchild);
        preorder(bt↑.rchild)
      END;
  END;
```

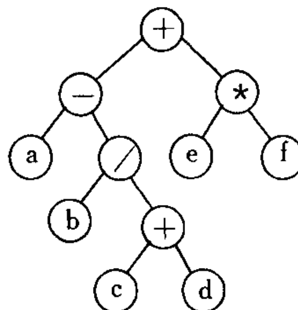


图 7-11　表达式 $a-b/(c+d)+e*f$ 的二叉树表示

例如，图 7-11 所示的二叉树表示下述表达式：

$$a-b/(c+d)+e*f$$

若先根遍历此二叉树，按访问结点的先后次序将结点排列起来，可得到二叉树的先序遍历序列（先序序列）即表达式的前缀表示为：$+-a/b+cd*ef$。

二、中根遍历

中根遍历又称中序遍历，其递归定义是：若二叉树为空，则空操作；否则，依次执行以下操作：

1. 中根遍历左子树；

2. 访问根结点；

3. 中根遍历右子树。

此递归过程的 PASCAL 语言描述如下：

```
PROCEDURE inorder(bt:bitreptr);
  BEGIN
    IF bt<>nil THEN
      BEGIN
        inorder(bt↑.lchild);
```

```
            write(bt ↑.data);
            inorder(bt ↑.rchild)
        END
    END;
```

对图 7-11 所示的二叉树，中根遍历得到的序列（中序序列）即表达式的中缀表示为：

$$a-b/c+d+e*f$$

下面我们讨论中根遍历的非递归算法。在非递归的算法中，需要用栈保存遍历所经的路径，以便在沿着结点的左指针遍历了它的左子树后，能退到该结点、并访问之。此算法的框图如图 7-12 所示。

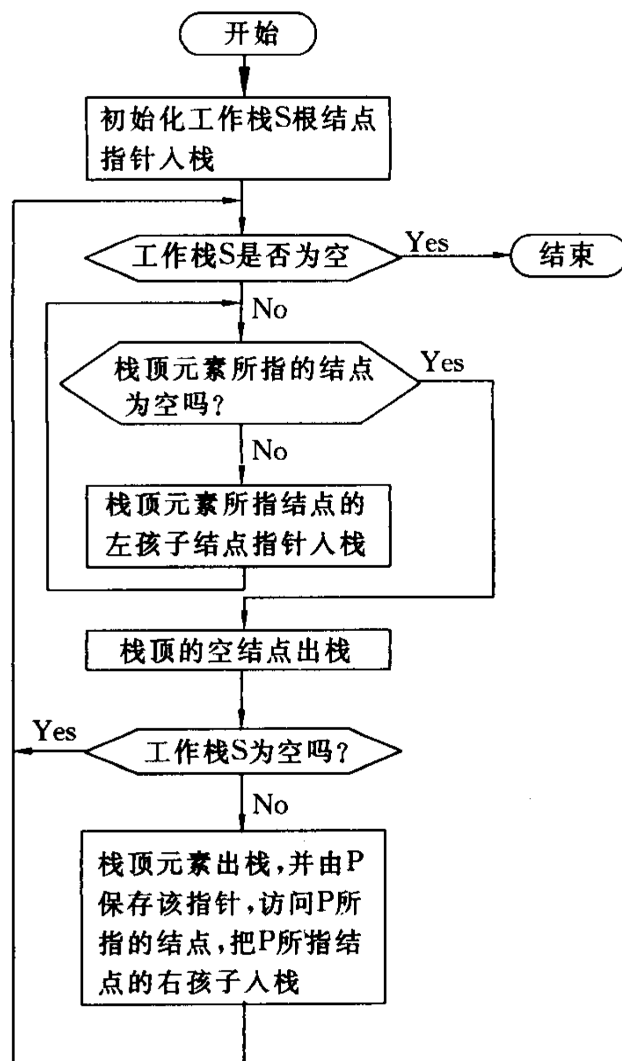


图 7-12 中根遍历的非递归算法框图

其 PASCAL 语言描述如下：

```
CONST smax={栈的最大容量}；
PROCEDER inorder(bt:bitreptr)；
VAR s:ARRAY[1..smax] OF bitreptr；
      p:bitreptr；
      top:integer；
BEGIN
  inistack(s)； {top:=0}
  push(s,bt)； {top:=top+1； s[top]:=bt}
  WHILE top>0 DO
    BEGIN
      WHILE s[top]<>nil DO
        BEGIN
          push(s,gettop(s)↑.lchild) {top:=top+1； s[top]:=s[top-1]↑.lchild}
        END；
      p:=pop(s)； {top:=top-1；}
      IF top>0 THEN
        BEGIN
          p:=pop(s)； {p:=s[top]； top:=top-1}
          write(p↑.data)；
          push(s,p↑.rchild) {top:=top+1；s[top]:=p↑.rchild}
        END
    END
END；
```

三、后根遍历

后根遍历又称后序遍历，其递归定义是：若二叉树为空，则空操作；否则，依次执行以下操作：

1. 后根遍历左子树；
2. 后根遍历右子树；
3. 访问根结点。

此递归过程的 PASCAL 语言描述如下：

```
PROCEDURE postorder(bt:bitreptr);
  BEGIN
    IF bt<>nil THEN
      BEGIN
        postorder(bt↑.lchild);
        postorder(bt↑.rchild);
        write(bt↑.data)
      END
  END;
```

对图 7-11 所示的二叉树，后根遍历得到的序列(后序序列)即表达式的后缀表示为：

abcd+/－ef＊+

对二叉树进行遍历的搜索路径除了上述按先序、中序和后序外，还可以按层次进行，即从上到下，从左到右地进行。

遍历二叉树的基本操作是访问结点。对含有 n 个结点的二叉树，上述遍历算法的时间复杂度均为 O(n)；所需辅助空间为遍历过程中栈的最大容量即二叉树的深度，最坏情况下为 n，所以空间复杂度也为 O(n)。

7.4 线索二叉树

遍历二叉树的算法给我们提供了将二叉树按一定次序把结点线性化的方法，使每个结点(除第一个和最后一个外)在这个线性序列中有且仅有一个直接前驱和直接后继。例如，在图 7-11 所示二叉树结点的中序序列 a－b/c+d+e＊f 中'c'的前驱为'/'，后继为'+'。

那么，对于一棵二叉树，如何求指定结点在任一序列中的前驱和后继呢？方法有多种。其一就是每一次都遍历一下二叉树。当然，这就需要花费一定的时间。其二就是通过一次遍历，并在遍历过程中保存这种信息。这就需要在每个结点上增加两个指针域，分别用于指向在某一遍历方案中该结点的前驱和后继。显然，这种方法降低了存储

密度。另外一种比较经济实惠的方法，就是下面要介绍的线索二叉树。

在前面介绍的二叉链表存储结构中，对有 n 个结点的二叉树有 n+1 个空指针域，由此设想能否利用这些空指针域来存放结点的前驱和后继信息。为了实现这个设想，我们对二叉链表中结点的指针域重新作如下规定：如果结点有左孩子，则其 lchild 域指向其左孩子结点；否则令 lchild 域指示其前驱。若结点有右孩子，则其 rchild 域指向其右孩子结点；否则令 rchild 域指示其后继。为了标识结点中两个指针域的指向，需要在结点中增加两个标志域 ltag 和 rtag，其结构如图 7-13 所示。

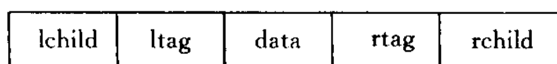


图 7-13　线索树的结点

其中：

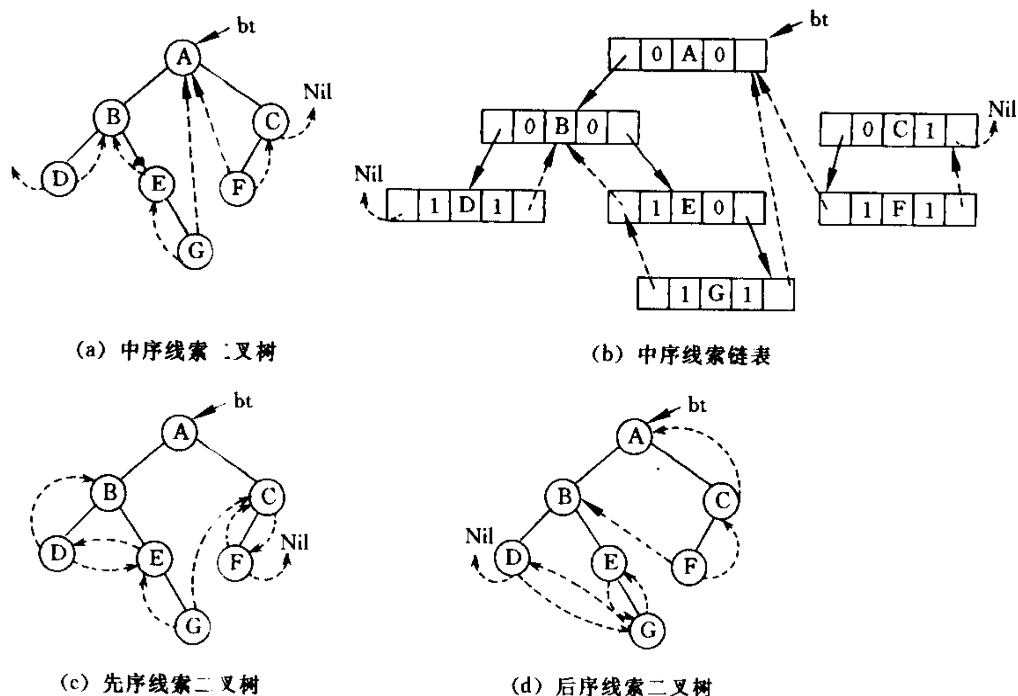
$$\text{ltag}=\begin{cases}0 & \text{lchild 指向结点的左孩子结点}\\ 1 & \text{lchild 指向结点的前驱结点}\end{cases}$$

$$\text{rtag}=\begin{cases}0 & \text{rchild 指向结点的右孩子结点}\\ 1 & \text{rchild 指向结点的后继结点}\end{cases}$$

以这种结点结构构成的二叉链表作为二叉树的存储结构，称为线索链表。其中指向结点前驱或后继的指针称为线索。带有线索的二叉树称为线索二叉树。例如，图 7-14(a)所示的中序线索二叉树，其对应的中序线索链表如图 7-14(b)所示。图 7-14(c)、(d)分别为先序线索二叉树和后序线索二叉树。其中实线为指针(指向左、右孩子)，虚线为线索(指向前驱或后继)。

一、建立线索二叉树

对二叉树以某种次序进行遍历并加上线索的过程称为线索化。那么，如何进行二叉树的线索化呢？由于线索化的实质是将二叉链表结点中的空指针改为指向其前驱或后继结点的线索；而前驱和后继的信息只有在遍历时才能得到，所以线索化的过程即为：在遍历过程中修改指针的过程。为了记下遍历过程中访问结点的先后次系，附



(a) 中序线索二叉树　(b) 中序线索链表

(c) 先序线索二叉树　(d) 后序线索二叉树

图 7-14　线索二叉树及其存储结构

设一个指针 pre，它用于指向当前访问结点的前驱。图 7-15 描述了通过中序遍历建立中序线索链表的算法框图，其中指针 p 指向当前访问的结点。

根据此框图，我们用 PASCAL 语言描述此算法。开始调用时，pre 为空，p 指向根结点。

```
TYPE thlinktp=↑thrnode;
     thrnode=RECORD
               data:integer;
               ltag,rtag:0..1;
               lchild, rchild:thlinktp
               END;
PROCEDURE inthread(p:thlinktp; VAR pre:thlinktp);
BEGIN
  IF p<>nil THEN
    BEGIN
      inthread (p↑.lchild, pre);
      IF p↑.lchild=nil THEN
        BEGIN
          p↑.ltag:=1; p↑.lchild:=pre
```

```
        END;
      IF pre↑.rchild=nil THEN
        BEGIN
          pre↑.rtag:=1; pre↑.rchild:=p
        END;
      pre:=p;
      inthread(p↑.rchild, pre)
    END
END;
```

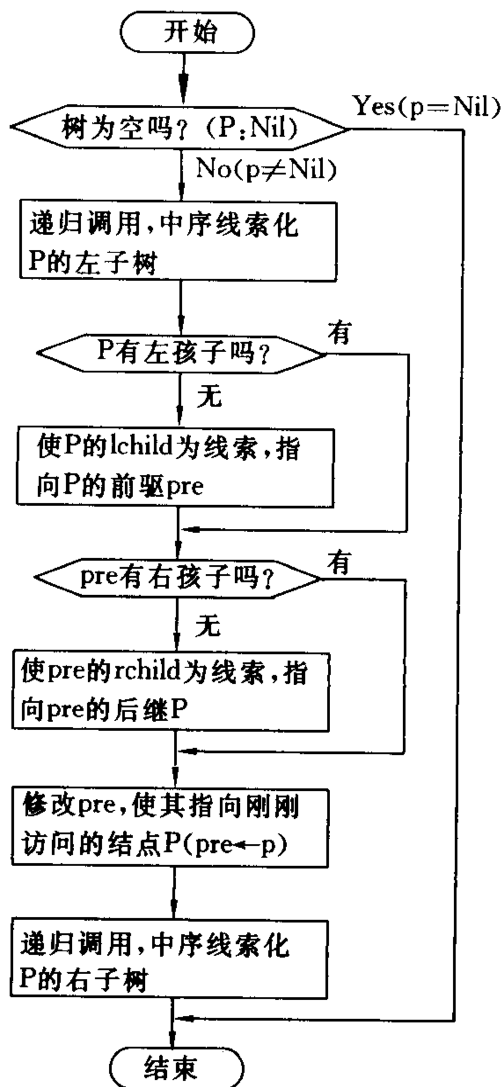


图 7-15　中序线索化算法框图

对于先序线索链表和后序线索链表的建立也可以通过先序遍历和后序遍历进行。请读者自己完成。

二、检索线索二叉树

在具有 n 个结点的线索二叉树中，只有 n+1 根线索(其余 2n-(n+1)=n-1 个是指针)，这还不足以直接指出任一结点的前驱和后继。那么如何来求任一指定结点的前驱和后继呢?

1. 在中序线索树上求指定结点的前驱和后继

在中序线索树上，求指定结点 p 的前驱，我们可以如下进行：如果 p 结点的左标志域为 1(p↑.ltag=1)，则说明 p 的 lchild 域为线索，即 p↑.lchild 指向 p 的前驱结点。例如，图 7-14(a)中，结点 F 的 lchild 域指向其前驱结点 A。如果 p↑.ltag=0，则说明 p 有左孩子，p 的前驱为其左子树上按中序遍历最后被访问的结点，即其左子树中最右下的结点。例如，在图 7-14(a)中，求 A 的前驱。首先根据 A 的 lchild 域找到其左子树的根 B；然后连续不断地沿着右链往下寻找，直至某一个结点的右标志域为 1，这个结点就是所求的前驱结点(在图中 A 结点的前驱就是 C)。在中序线索树上求指定结点 p 的前驱的算法框图如图 7-16(a)所示。

在中序结索树上求指定结点 p 的后继，同求 p 的前驱类似：如果 p↑.rtag=1，则 p 的 rchild 域为线索，即 p↑.rchild 指向其后继；否则(p↑.rtag=0)p 有右孩子，p 的后继为其右子树上按中序遍历最先被访问的结点，即右子树上最左下的结点。其算法框图如图 7-16(b)。

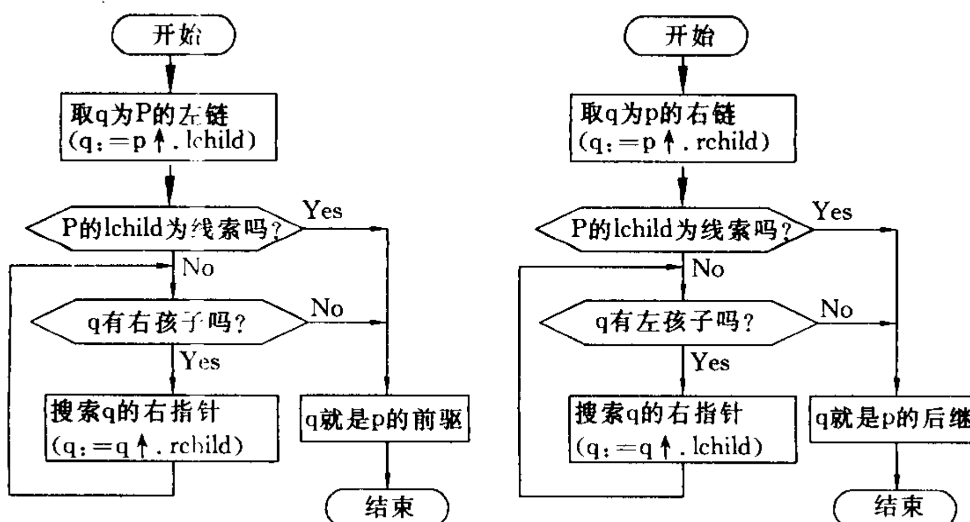
2. 在后序线索树上求指定结点的前驱和后继

在后序线索树上求指定结点 p 的前驱比较简单。如果 p 有右孩子(即 p↑.rtag=0)，则 p 的前驱就是其右孩子；否则，也就是 p 没有右孩子，则不管 p 有无左孩子，p 的 lchild 域指的总是其前驱。

在后序线索树上求指定结点 p 的后继比较麻烦。如果 p 的右标志为 1，则 p 的 rchild 指向其后继；否则需要先求出 p 的双亲(设用 f 指示)，然后：

(1) 若 f 无右孩子或 p 是 f 的右孩子，则 p 的后继就为其双亲 f。

(2) 若 p 为 f 的左孩子，并且 f 另外有右孩子 q，则 p 的后继为 f



(a) 中序线索树上求指定结点 P 的前驱　　(b) 中序线索树上求指定结点 P 的后继

图 7.16　在中序线索树上求指定结点 p 的前驱和后继

的右子树上按后序遍历最先访问的结点。这个结点我们可以按下面的方法寻找:沿 f 的右孩子 q 的左链搜索,直至某一个无左孩子的结点;再看此结点是否有右孩子,若有,则沿此右孩子的左链搜索。如此重复,直至找到某一叶子结点,此叶子结点就是 p 的后继。可见,在后继线索树上找后继时需要知道结点的双亲。

另外,在先序线索树上求指定结点的后继比较简单,类似于在后序线索树上求前驱。而求指定结点的前驱却比较麻烦,其算法与在后序线索树上求指定结点的后继十分类似,这里不再详述。

三、在线索树上进行插入操作

一般情况下,在线索二叉树上插入或删除一棵子树,就会破坏线索。所以,在修改结点指向孩子结点的指针时,还必须修复线索。下面我们仅讨论在中序线索上插入一棵子树的操作。设指针 p 指向待插入的中序线索树的根结点且 p 结点无右子树,指针 q 指向被插中序线索树 bt 中的某一结点。操作 addlchild(bt,q,p)是将以 p 为根的中序线索树插入到中序线索树 bt 中,成为 q 的左子树;如果 q 原来有左子树,则在插入之后成为 p 的右子树。这个操作可分二种情况进行。

设指针 s 指向以 p 为根的中序线索树中第一个被中序访问的结点，

1. 若 q 结点原来没有左子树，则按中序遍历规则，插入后原来 q 的前驱变为 s 的前驱，且 p 的后继为 q。图 7-17(a) 表示了这种插入操作前后线索二叉树的情形。

2. 若 q 结点原来有左子树，设指针 t 指向 q 的左子树中第一个被中序访问的结点，则插入后原来 t 的前驱成为 s 的前驱，而 p 成为 t 的前驱 。图 7-17(b)表示了这种插入操作前后线索二叉树的情形。

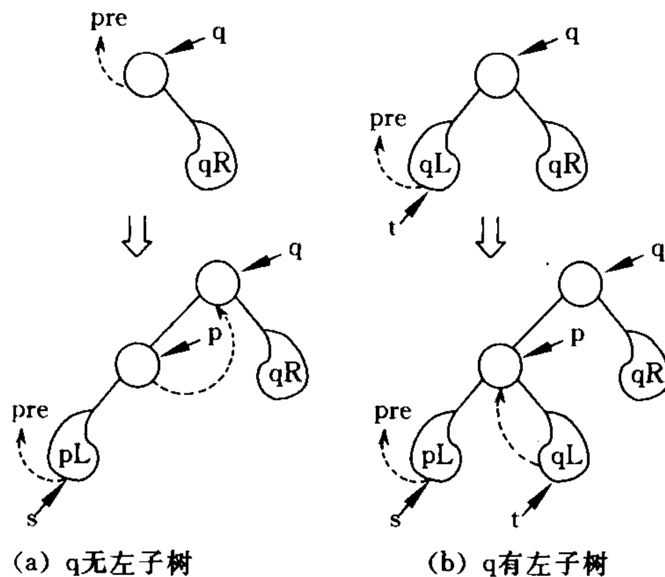


图 7-17　中序线索树上插入左子树示例

这个操作的算法，可由图 7-18 的框图描述：

其 PASCAL 语言描述如下：

```
PROCEDURE addlchild(bt,q,p:thlinktp);
VAR s,t:thlinktp;
BEGIN
  s:=p;
  WHILE s↑.ltag=0 DO s:=s↑.lchild;
  IF q↑.ltag=0 THEN
    BEGIN
      t:=q↑.lchild;
      WHILE t↑.ltag=0 DO t:=t↑.lchild;
```

```
          s↑.lchild:=t↑.lchild;
          t↑.lchild:=p;
          p↑.rtag:=0; p↑.rchild:=q↑.lchild;
          q↑.lchild:=p
        END
    ELSE BEGIN
          s↑.lchild:=q↑.lchild;
          p↑.rchild:=q;
          q↑.ltag:=0; q↑.lchild:=p
    END
END;
```

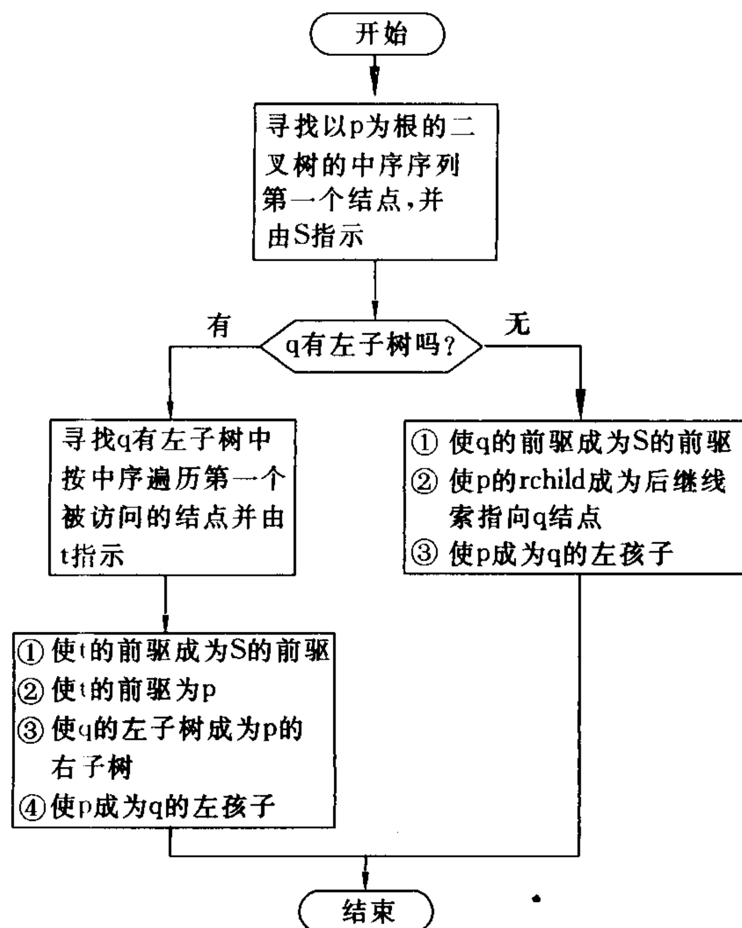


图 7-18　中序线索树上插入左子树的框图

7.5 二叉树与树和森林之间的转换

一、二叉树与树之间的转换

在这里我们为了不引起混淆，规定一般树当作有序树来处理。

1. 一般树转化为二叉树

将一般树转换成二叉树可按下列步骤进行操作：

(1) 加线：在各相邻的兄弟之间加一条连线。

(2) 抹线：对每个结点，除了其最左的一个孩子以外，抹掉该结点原先与其余孩子之间的连线。

(3) 旋转：将图形作适当的旋转，使之成为二叉树的树形结构。具体做法是：新加上去的连线均向右下斜，原来的连线均向左斜。

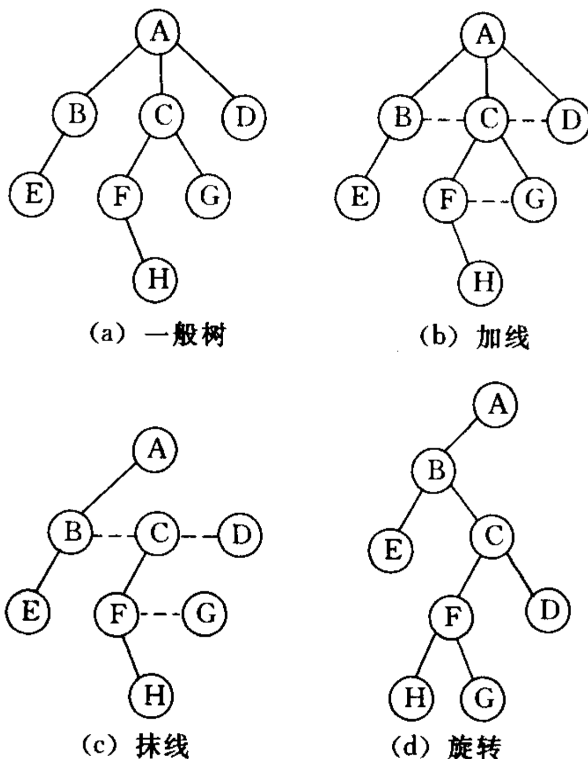


图 7-19 一般树转化为二叉树的过程

图 7-19 展示了一般树转化为二叉树的过程。

从中我们可以看出，转换后的二叉树根结点没有右子树，其余结点的右孩子是原来树中和该结点相邻的兄弟，结点的左孩子是原来树中该结点的最左孩子。

2. 二叉树转化为一般树

二叉树转化为一般树时，要求二叉树的根结点没有右子树，转化

的具体步骤如下：

(1) 加线：对二叉树中的每一个结点i，若结点i是其双亲的左孩子，则将该结点的右孩子以及沿着此右孩子的右链连续不断搜索到的所有右孩子，都分别与结点i的双亲用线连起来。如图7-20(b)所示。

(2) 抹线：抹掉原二叉树中所有双亲结点与右孩子之间的连线。

(3) 美化：让各结点按层次排列，使之成为树形结构。

图7-20展示了二叉树转化为一般树的过程。

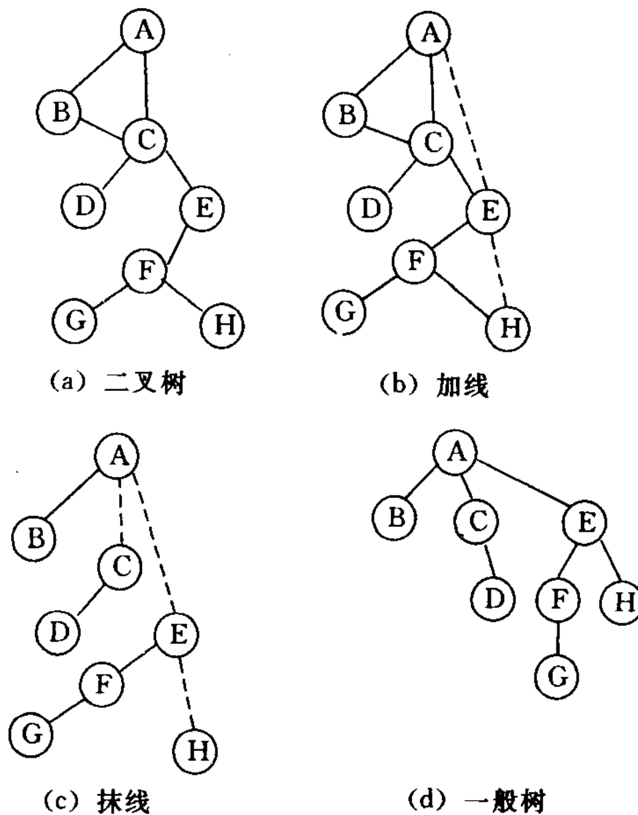


图7-20　二叉树转化为一般树的过程

二、二叉树与森林之间的转化

1. 森林转化为二叉树

森林是树的有限集合，所以我们可以按如下方法把它转换成二叉树：

(1) 首先将各棵树分别转换为二叉树。

(2) 然后再按森林中树的次序，依次将后一棵二叉树作为前一棵二叉树根结点的右子树。这样，第一棵树的根就是转化后二叉树的根。

图 7-21 为森林转化为二叉树的示例。

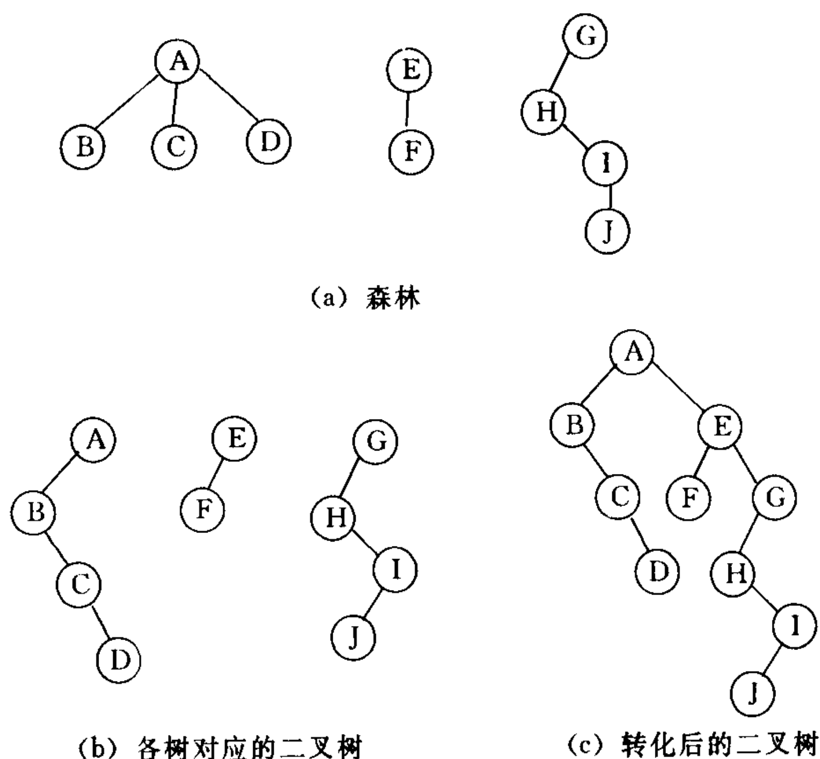


图 7-21　森林转化为二叉树的图示

2. 二叉树转化为森林

将一棵二叉树转化成森林，可按如下步骤进行：

(1) 抹线：将二叉树根结点与其右孩子之间的连线，以及沿着此右孩子的右链连续不断搜索到的右孩子间的连线抹掉。这样就得到了若干棵根结点没有右子树的二叉树。

(2) 将得到的这些二叉树用前述方法分别转化成一般树。

三、树的遍历

对于树我们可以定义如下二种次序的遍历：

1. 先根(先序)遍历树

先访问树的根结点，然后依次先根遍历根的每棵子树。例如，对

图 7-19(a)所示的树，其先根遍历所得的先根序列为：A、B、C、F、H、G、D。

2. 后根(后序)遍历树

先依次后根遍历根的每棵子树，然后再访问根结点。例如，对图 7-19(a)所示的树，其后根遍历所得的后根序列为：E、B、H、F、C、D、A。

从前面树与二叉树之间的转换关系可以得到：树的先根遍历和后根遍历可分别借助于对应二叉树的先根遍历和中根遍历完成。

四、森林的遍历

对于森林我们也可以定义如下两种遍历方法：

1. 先序遍历森林

若森林非空，则可按如下步骤遍历森林：

(1) 访问森林中第一棵树的根结点；

(2) 先序遍历第一棵树中根的子树森林；

(3) 先序遍历除第一棵树以外其余树组成的森林。

2. 中序遍历森林

若森林非空，则可按如下步骤遍历森林：

(1) 中序遍历森林中第一棵树的根结点的子树森林；

(2) 访问第一棵树的根结点；

(3) 中序遍历除去第一棵树之后剩余的树构成的森林。

例如，对图 7-21(a)所示的森林进行先序遍历和中序遍历，可得到森林的先序序列为：A、B、C、E、F、G、H、I、J；中序序列为：B、C、D、A、F、E、H、J、I、G。

从前面森林与二叉树之间转换关系可知，当森林转化为二叉树后，其第一棵树的根的子树森林成为二叉树根的左子树，除第一棵以外的其它树构成的森林成为二叉树根的右子树。所以，森林的先序遍历和中序遍历即为其对应二叉树的先序遍历和中序遍历。

7.6 哈夫曼树及其应用

一、基本概念

在讨论哈夫曼树之前，首先介绍有关树的路径长度概念。在此定义：从树中一个结点到另一个结点之间的分支构成这两个结点之间的路径。路径上分支的数目称为路径长度。树的路径长度是从根结点到所有结点的路径长度之和。例如，图 7-22(a) 所示，含有 8 个结点的二叉树，其路径长度为 14。显然我们可以看出：在具有 n 个结点的二叉树中，完全二叉树具有最小的路径长度，但是具有最小路径长度的不一定是完全二叉树。例如，图 7-22(b)、(c)所示的含有 8 个结点的二棵不同的二叉树，它们都具有最小的路径长度，但图 7-22(c) 不是完全二叉树。

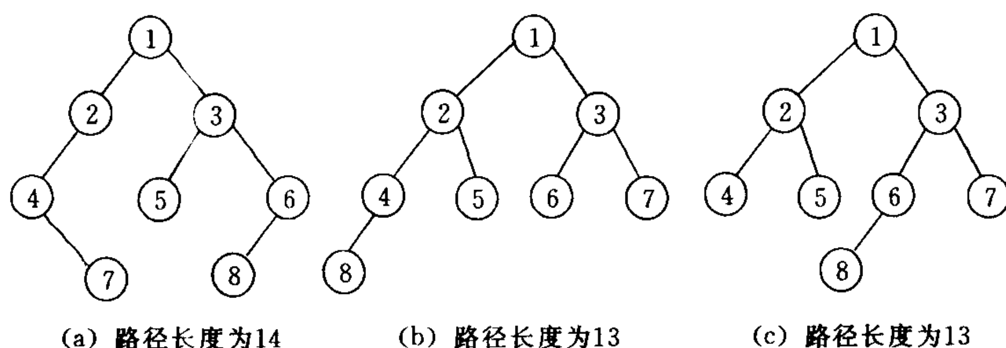


图 7-22　二叉树的路径长度

下面我们将上述概念进行推广，考虑带权的情况。结点的带权路径为从树根到该结点的路径长度与结点上权的乘积。树的带权路径长度为树中所有带权叶子结点的带权路径长度之和，通常记为 WPL。假设有 n 个权值为$\{W_1, W_2, \cdots, W_n\}$，构造一棵含有 n 个叶子结点的二叉树，叶子的权分别为 $W_i(i=1, \cdots, n)$，则其中 WPL 最小的二叉树称为最优二叉树或哈夫曼树。例如，图 7-23 中的三棵二叉树，都有 4 个叶子结点，且它们的权分别为 10、7、5、3。这三棵二叉树的带权路径长度分别为：

(a) WPL = 10 * 2 + 7 * 2 + 5 * 2 + 3 * 2 = 50

(b) WPL = 10 * 3 + 7 * 3 + 5 * 2 + 3 * 1 = 64

(c) WPL = 3 * 3 + 5 * 3 + 7 * 2 + 10 * 1 = 48

其中(c)的 WPL 最小,可以验证,它是一棵哈夫曼树。

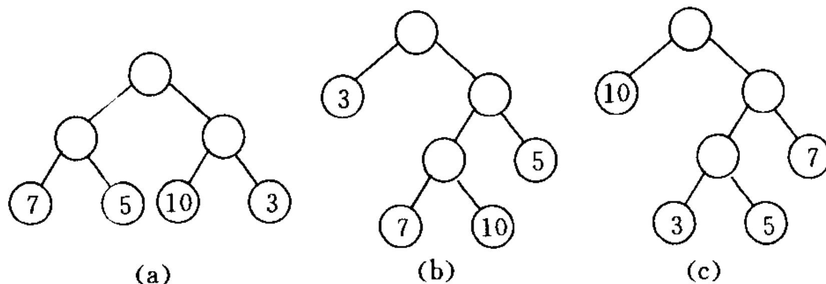


图 7-23　叶子带权相同的三棵二叉树

显然,带权路径长度最小的二叉树不一定是完全二叉树;但当 Wi(i=1,…,n)相等时，完全二叉树一定是哈夫曼树。那么，对于给定的 n 个叶子结点的权 Wi(i=1,…,n)，如何构造出一棵哈夫曼树呢？下面我们就讨论哈夫曼树的构造算法。

二、哈夫曼算法

哈夫曼最早给出了构造具有最小带权路径长度的二叉树的算法,俗称哈夫曼算法,其设计思想如下所述:

(1) 根椐给定的 n 个权值{W_1,W_2,…,W_n},构造一个森林 F,其中每棵二叉树 T_i 只有一个权为 W_i 的结点。

(2) 在 F 中选取两棵根结点的权值最小的二叉树 T_i 和 T_j,生成一棵新的二叉树,其根结点的左右子树分别为 T_i、T_j,且根结点的权为其左右子树根结点的权值之和。

(3) 在 F 中删除 T_i、T_j,并把新生成的二叉树加到 F 中。

(4) 重复(2)、(3),直到 F 中只有一棵二叉树,它就是哈夫曼树。

图 7-24 展示了一棵哈夫曼树的构造过程。

哈夫曼树在信息检索和通讯编码中很有用,下面我们讨论这个算法的具体实现。

由于哈夫曼树中没有度为 1 的结点,所以一棵具有 n 个叶子的

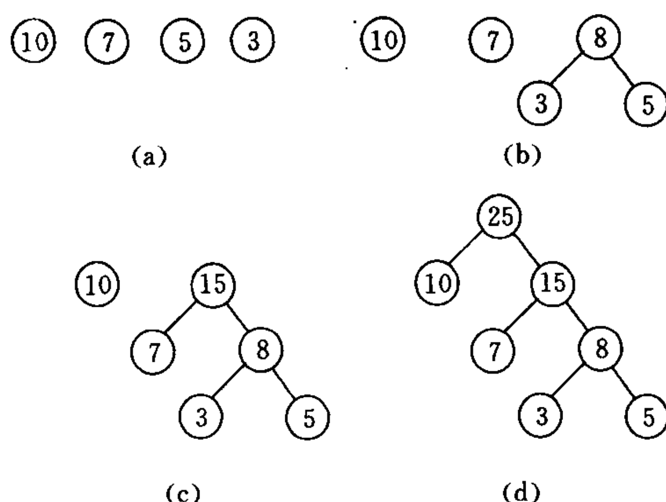


图 7-24　哈夫曼树的构造过程

哈夫曼树共有 2n－1 个结点。我们可以用一个大小为 2n－1 的向量表示哈夫曼树，每个分量表示哈夫曼树中的一个结点，它的结构如下：

weight	lch	parent	rch

其中：weight 用于存放结点的权值。lch、rch 分别用于指示该结点的左右孩子结点在向量中的序号；当 lch 和 rch 为 0 时，表示该结点为叶子结点。parent 用于指示该结点的双亲在向量中的序号；当 parent 为 0 时，表示该结点为一棵子树的根。图 7-25 为哈夫曼算法的框图描述。

其 PASCAL 语言描述如下：

```
        CONST n={叶子数目的最大值}；
              m=2＊n－1；
        TYPE nodetype=RECORD
                          weight:integer;
                          parent,lch,rch:0..m
                        END;
             hufftree=ARRAY [1..m] OF nodetype;
          PROCEDURE huffman (w: ARRAY [1.. n] OF integer; var ht:
hufftree);
        VAR i,t1,t2,w1,w2,j:integer;
```

```
BEGIN
  FOR i:=1 TO m DO
    BEGIN
      ht[i].parent:=0; ht[i].lch:=0; ht[i].rch:=0
    END;
  FOR i:=1 TO n DO ht[i].weight:=w[i];
  i:=0;
  WHILE i<n-1 DO
    BEGIN
      w1:=maxint; w2:=maxint;t1:=0;
      FOR j:=1 TO n+i DO
        IF ht[j].parent=0 THEN
          IF ht[j].weight<w1 THEN
            BEGIN
              w2:=w1; t2:=t1;
              w1:=ht[j].weight; t1:=j
            END
          ELSE IF ht[j].weight<w2 THEN
                BEGIN
                  w2:=ht[j].weight;
                  t2:=j
                END;
          i:=i+1;
          ht[t1].parent:=n+i; ht[t2].parent:=n+i;
          ht[n+i].lch:=t1; ht[n+i].rch:=t2;
          ht[n+i].weight:=w1+w2
    END
END;
```

三、哈夫曼树的应用

哈夫曼树的应用很广。在不同的应用中，叶子的权值可以有不同的解释。当哈夫曼树应用于信息编码，则一个叶子代表一个信息符

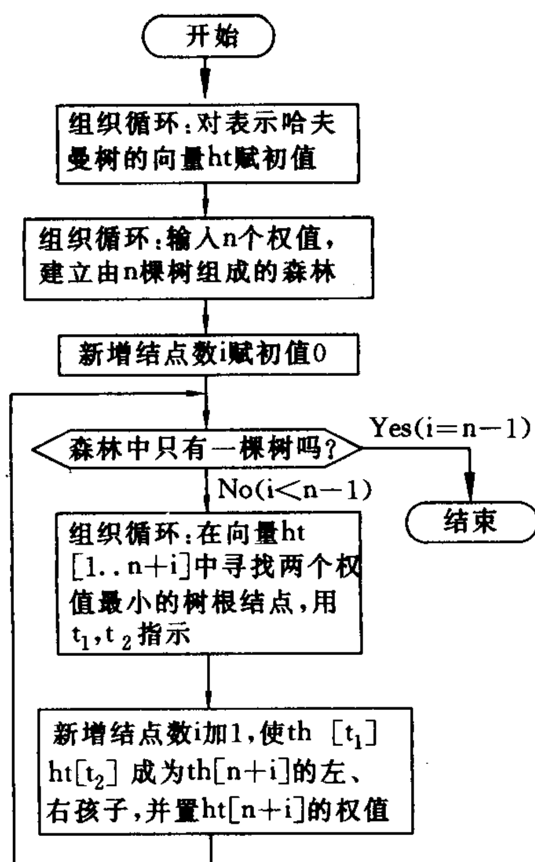


图 7-25 构造哈夫曼树的算法框图

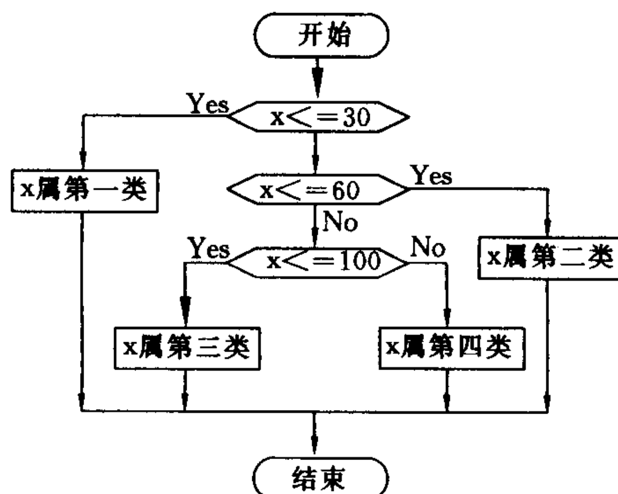
号,其权值就是此符号出现在编码中的频率;当应用于判定过程,则一个叶子代表一类数据,其权值就是此类数据出现的频率。下面我们将分别介绍。

1. 哈夫曼树在判定过程中的应用

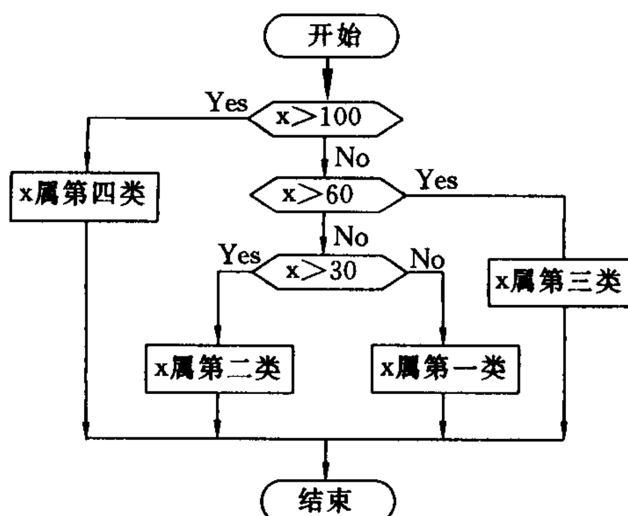
种用哈夫曼树可以构成最佳判定过程。判定过程可以用二叉树描述,从树根开始进行测试,测试的结果分成二个分支,选其中的一个分支再进行测试,……。例如,要对一批正整数进行分类:x<=30时属第一类,30<x<=60 时属第二类,60<x<=100 时属第三类,x>100 时属第四类。图 7-26(a)为判定一个数 x 属哪一类的算法框图。

假设 x 属第一、二、三、四类的概率分别为:

x	第一类	第二类	第三类	第四类
概率	0.1	0.2	0.3	0.4



(a)判定过程之一



(b)判定过程之二

图 7-26　哈夫曼树在判定中的应用

此时，70%的数据需进行三次判别，20%的数据需进行二次判别，只有 10%的数据需进行一次判别。如果以 10、20、30、40 为权构造一棵哈夫曼树，则可得图 7-26(b)所示的判定过程，它可使 40%的数据只需一次测试，30%的数据只需二次测试，还有 30%的数据

需三次测试。显然,(b)要比(a)的平均比较次数少。

2. 哈夫曼编码

在进行电报通讯时,需要将传递的文字转换成由二进制数字组成的串。例如,假设需传送的电文为'ABACCDA',它只有四种字符,所以只需用两位二进制数字便可区分。设A、B、C、D的编码分别为00、01、10、11,则上述七个字符的电文便为'00010010101100'共14位二进制数字。对方接到后,便可按二位一分进行译码。

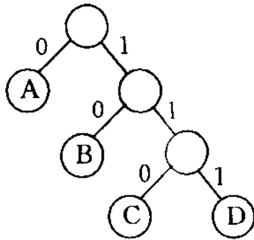


图 7-27 前缀码示例

然而在传送电文时,我们总希望电文越短越好。如果对每个文字设计长度不等的编码,且让在电文中出现次数较多的文字,采用尽可能短的编码,则传送的电文总长便可减小。例如,设A、B、C、D的编码分别为0、00、1和01,则上述七个字符组成的电文便可用长度为9的二进制串'000011010'表示。但是,这样的电文无法正确翻译。例如,传送的二进制串的前四个数字'0000'就可有多种译法:或为'AAAA';或为'BAA';或为'BB'等。为此,在使用不等长编码时,任一个字符的编码必须不是其他字符编码的前缀,这种编码称为前缀码。我们可以利用二叉树来设计二进制的前缀码。例如,图7-27所示的二叉树,其四个叶子分别表示A、B、C、D四个字符。若约定左分支为二进制数字'0',右分支为二进制数字'1',则可以把从根结点到叶子结点的路径上分支数字组成的二进制串作为该叶子结点所代表的字符编码。显然,如此得到的一定是二进制前缀码。在图7-27中A、B、C、D的前缀码分别为0、10、110、111,则上述七个字符组成的电文便为'01001101101110',但它不是最短的。

现在的问题是如何得到电文总长度最短的二进制前缀码呢?

假设有n种字符$(c_1, c_2, \cdots, c_n)$,字符c_i在电文中出现的次数为W_i,其编码长度为L_i,则由这n种字符组成的电文,其二进制编码串的总长度为$W_1 * L_1 + W_2 * L_2 + \cdots + W_n * L_n$。对应于二叉树,设$W_i$为叶子字符的权值,则Li恰为从根到该叶子的路径长度,$W_1 * L_1$

$+W_2*L_2+\cdots+W_n*L_n$ 恰为二叉树的带权路径长度。由此可见，设计电文总长度最短的二进制前缀码的过程，即为以 n 种字符出现的频率作为叶子字符的权，设计一棵哈夫曼树的过程。由此得到的二进制前缀码称为哈夫曼编码。例如：设 A、B、C、D 在电文中出现的频率分别为 0.5、0.1、0.35、0.05，则按权{50,10,35,5}构造的哈夫曼树如图 7-28 所示。这样，A、B、C、D 的哈夫曼编码分别为 0、101、11 和 100。

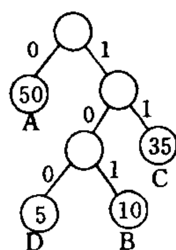


图 7-28　哈夫曼编码

习　题

1. 设二叉树结点的先序序列为 A、B、C，问有几种不同的二叉树可以得到这一遍历结果？并把它们画出来。
2. 设一棵度为 m 的树中有 n_1 个度为 1 的结点，n_2 个度为 2 的结点，…，n_m 个度为 m 的结点，问该树中有多少叶子结点？
3. 试以二叉链表作存储结构，编写算法将二叉树中所有结点的左、右子树相互交换。
4. 试以二叉链表作存储结构，编写将二叉树按层次顺序（同一层次自左至右）遍历的算法。
5. 试以二叉链表作存储结构，编写计算二叉树中叶子结点总数的算法。
6. 设二叉树结点的先序序列和中序序列分别为：A,B,C,D,E,F,G,H,I 和 B,C,A,E,D,G,H,F,I，试画出该二叉树，并对该二叉树进行后序线索化。
7. 给定如下一组权值{4，2，3，5，7，8}，建立一棵哈夫曼树。

第八日　图

图是一种比线性表和树更为复杂的数据结构。在线性表中，每个数据元素最多有一个直接前驱和一个直接后继；在树中，数据元素之间有着明显的层次关系，并且每一层上的数据元素可以和它下一层中的多个元素相联系，但只能和它上一层中的一个元素相联系；而在图中，任意两个数据元素之间都可以有联系。

图的应用十分广泛，在"离散数学"课程中有专门关于图的理论研究。在此，我们仅讨论应用图论的知识如何实现图的操作，包括图的存储结构及其应用。

8.1　图的基本概念和基本操作

一、图的定义和术语

图是一种数据结构，它由两个集合 V(G)和 E(G)组成，记为 G＝(V,E)。其中，V(G)为图 G 中数据元素（称为顶点）的非空有限集；E(G)是图 G 中顶点之间的关系的集合。

在图 G 中，如果顶点之间的关系是有序对，则称 G 为有向图。顶点之间关系的有序对$\langle x,y\rangle \in E$、$(x,y\in V)$称为从顶点 x 到 y 的一条弧，且 x 称为弧尾、y 称为弧头。例如，图 8-1(a)所示为一个有向图 G_1，其中：

$$V(G_1)=\{1,2,3\};E(G_1)=\{\langle 1,2\rangle,\langle 2,1\rangle,\langle 2,3\rangle\}$$

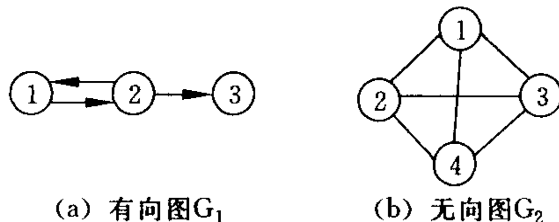
在这里$\langle 1,2\rangle$和$\langle 2,1\rangle$是两条不同的弧。

在图 G 中，如果顶点之间的关系是无序对，则称 G 为无向图。顶

点之间关系的无序对(x,y)∈E、(x,y∈V)称为 x 和 y 之间的一条边。例如,图 8-1(b)所示图 G_2 是无向图。其中:

$V(G_2)=\{1,2,3,4\}$;

$E(G_2)=\{(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)\}$



(a) 有向图G_1　　(b) 无向图G_2

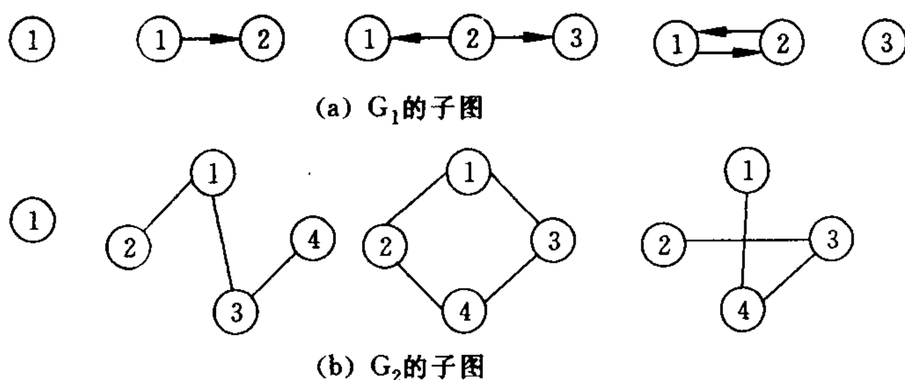
图 8-1　图的示例

在下面的讨论中,我们不考虑顶点到其自身的弧或边,即如果弧 $\langle v_i,v_j\rangle \in E(G)$ 或边 $(v_i,v_j)\in E(G)$ 时,$i\neq j$。在这里,我们用 n 表示图中顶点的数目,用 e 表示弧或边的数目。那么对于无向图,e 的取值范围从 0 到 $n(n-1)/2$,并且称具有 $n(n-1)/2$ 条边的无向图为完全图;对于有向图,e 的取值范围从 0 到 $n(n-1)$,并且称具有 $n(n-1)$条边的有向图为有向完全图。当图中的边或弧很少(如 $e<n\log n$)时,则称其为稀疏图,反之称为稠密图。

有时图的边或弧具有与之相关的数称为权,这些权可以表示从一个顶点到另一个顶点的距离或时间等。这时,我们称这种带权的图为网。

设有两个图 $G=(V,E)$ 和 $G'=(V',E')$,如果 $V'\subseteq V$ 且 $E'\subseteq E$,则称 G' 为 G 的子图。例如,图 8-2(a)是图 8-1(a)的一些子图,图 8-2(b)是图 8-1(b)的一些子图。

对于无向图 $G=(V,E)$,如果边 $(v_i,v_j)\in E$,则称顶点 v_i 和 v_j 互为邻接点,即 v_i 和 v_j 相邻接;称边 (v_i,v_j) 依附于顶点 v_i 和 v_j,或者说边 (v_i,v_j) 与顶点 v_i、v_j 相关联;称与顶点 v 相关联的边的数目为 v 的度,记为 TD(v)。例如,G_2 中顶点 v_3 的度为 3。对于有向图 $G=(V,E)$,如果弧 $\langle v_i,v_j\rangle\in E$,则称顶点 v_i 邻接到顶点 v_j,顶点 v_j 邻接自顶点 v_i;称弧 $\langle v_i,v_j\rangle$ 与顶点 v_i、v_j 相关联。以顶点 v 为头的弧的数目称为 v 的入度,记为 ID(v);以顶点 v 为尾的弧的数目称为 v 的出度,记为



(a) G_1的子图

(b) G_2的子图

图 8-2 子图的示例

OD(v)；顶点 v 的度 TD(v)=ID(v)+OD(v)。例如，G_1 中顶点 v_2 的出度为 2，入度为 1，度为 3。一般地，若图 G 有 n 个顶点，e 条边或弧，则有：

$$e=(TD(v_1)+TD(v_2)+\cdots+TD(v_n))/2$$

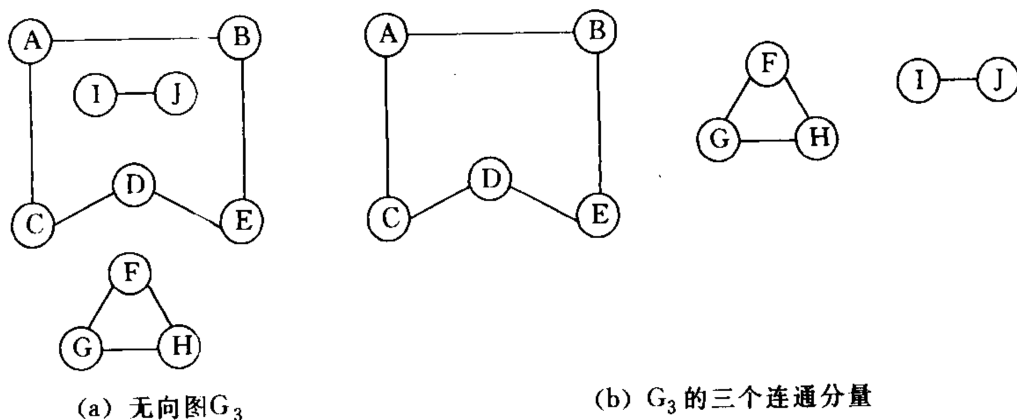
在无向图 G=(V,E)中，从顶点 v_p 到顶点 v_q 的一条路径是一个顶点序列(v_p，v_{i1}，v_{i2}，…，v_{in}，v_q)且(v_p，v_{i1})、(v_{i1}，v_{i2})、…、(v_{in}，v_q)是 E(G)中的边，路径上边的数目称为该路径的长度。例如，在图 8-1(b)的 G_2 中，从顶点 v_1 到 v_3 可以通过边(v_1，v_4)，(v_4，v_2)，(v_2，v_3)而到达，其路径为(v_1，v_4，v_2，v_3)，路径长度为 3。

对于有向图，路径由弧组成、因而是有向的。例如，在图 8-1(a)的 G_1 中，从 v_1 到 v_3 的路径可通过弧$\langle v_1,v_2\rangle$，$\langle v_2,v_3\rangle$而到达，其路径可记为$\langle v_1,v_2,v_3\rangle$，路径长度为 2。

在一条路径中，顶点不重复出现的路径称为简单路径；第一个顶点和最后一个顶点相同的路径称为回路或环；除第一个顶点和最后一个顶点外，其余顶点都不相同的回路称为简单回路。

在无向图中，若从 v_i 到 v_j 有路径存在，则称 v_i 和 v_j 是连通的。如果在图 G 中，任何两个顶点都是连通的，则称 G 为连通图。无向图中的极大连通子图称为它的连通分量。例如，图 8-1 中的 G_2 为连通图；图 8-3(a)中的 G_3 不是连通图，它的三个连通分量如图 8-3(b)所示：

在有向图 G 中，如果对于一对顶点 v_i，$v_j\in V$ 且 $v_i\neq v_j$，从 v_i 到 v_j 和 v_j 到 v_i 都存在路径，则称 v_i 和 v_j 是强连通的。如果在图 G 中，任



(a) 无向图G_3　　(b) G_3的三个连通分量

图 8-3　无向图及其连通分量

何两个顶点都是强连通的，则称 G 是强连通图。有向图中的极大强连通子图称为它的强连通分量。例如，图 8-1(a)中的 G_1 不是强连通图，它有二个强连通分量，如图 8-4 所示：



图 8-4　G_1 的强连通分量

一个连通图的生成树是该连通图的一个极小连通子图，它含有图中全部 n 个顶点，但是只有足以构成一棵树的n－1条边。例如，图 8-5 是 G_2 的一棵生成树。如果在一棵生成树上添加一条边，则必定构成一个环，因为这条边使得它所依附的那两个顶点之间有了第二条路径。

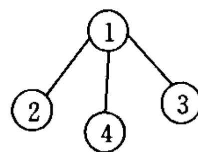


图 8-5　G_2 的生成树

二、图的基本操作

1. 顶点定位函数(loc-vextex(G,v))：确定顶点 v 在图 G 中的位置。若图中无此顶点，则函数值为“零”。
2. 取顶点函数(get-vextex(G,i))：求图 G 中第 i 个顶点。若 i 大于图 G 中的顶点数，则函数值为“空”。
3. 求第一个邻接点函数(first-adj(G,v))：求图 G 中顶点 v 的第一个邻接点。若 v 没有邻接点或图 G 中无顶点 v，则函数值为“空”。
4. 求下一个邻接点函数(next-adj(G,v,w))：已知 w 为图 G 中顶点 v 的一个邻接点，求顶点 v 的 w 以下一个邻接点，若 w

是 v 的最后一个邻接点，则函数值为“空”。

5. 插入顶点(ins-vextex(G,v))：在图 G 中增加一个顶点 v，使之成为图 G 的第 n+1 个顶点，其中 n 为插入之前图 G 中的顶点数。
6. 删除顶点(del-vextex(G,v))：在图 G 中删除顶点 v 以及所有与顶点 v 相关联的弧。
7. 插入弧(ins-arc(G,v,w))：在图 G 中增加一条从顶点 v 到 w 的弧。
8. 删除弧(del-arc(G,v,w))：在图 G 中删除一条从顶点 v 到 w 的弧。

8.2 图的存储结构

图的结构比较复杂，应用也很广泛，所以图的存储结构也比较多。对图的存储结构的选择取决于具体应用所需进行的操作。下面介绍四种最常用的存储结构：邻接矩阵、邻接表、十字链表和邻接多重表。

一、邻接矩阵

如果图中顶点除了编号外别无其它信息，则可采用邻接矩阵存储结构。图的邻接矩阵存储结构是用一个二维数组来表示图中顶点以及顶点间的相邻关系。设图 G 中有 n>=1 个顶点，则 G 的邻接矩阵是按如下定义的一个 n 阶方阵 A：

$$A[i,j]=\begin{cases}1,\text{若}(v_i,v_j)\text{或}\langle v_i,v_j\rangle\in E(G)\\0,\text{反之}\end{cases}$$

例如，图 8-1 中的 G_1,G_2 的邻接矩阵分别表示为 A_1 和 A_2，矩阵中的行、列号对应于图中顶点的编号。

$$A_1=\begin{bmatrix}0&1&0\\1&0&1\\0&0&0\end{bmatrix}\qquad A_2=\begin{bmatrix}0&1&1&1\\1&0&1&1\\1&1&0&1\\1&1&1&0\end{bmatrix}$$

显然，无向图的邻接矩阵是对称的，故对具有 n 个顶点的无向图，只需存放它的下(或上)三角矩阵，仅占 $n(n+1)/2$ 个单位的存储空间；用邻接矩阵来表示一个具有 n 个顶点的有向图时则需要 n^2 个单位的存储空间。

在图的邻接矩阵上，很容易判定图中任意两个顶点之间是否相邻接，也很容易求各个顶点的度数。对于无向图，邻接矩阵第 i 行元素之和就是图中第 i 个顶点的度数；对于有向图，邻接矩阵第 i 行元素之和为顶点 i 的出度，第 i 列元素之和为顶点 i 的入度。

二、邻接表

邻接表是图的一种链式存储结构。在邻接表中，为图中每个顶点建立一个单链表，第 i 个单链表中的结点表示依附于顶点 v_i 的边(对有向图表示以顶点 v_i 为尾的弧)。链表中的每个结点由二个域组成，如图 8-6(a)所示。其中，邻接点域(adjvex)指示与顶点 v_i 邻接的顶点在图中的位置，链域(nextare)用于指示另一条依附于 v_i 的边或另一条以 v_i 为尾的弧。每个链表附设一个表头结点，表头结点由二个域组成，如图 8-6(b)所示。其中，数据域(vexdata)用于存储顶点的名或有关顶点的其它信息，链域(firarc)用于指向链表中的第一个结点；并且这些表头结点可以用一个向量存储，以便能随机访问任一个顶点的链表。例如，图 8-6(c)和(d)所示分别为图 8-1 中图 G_1 和图 G_2 的邻接表。

一个含有 n 个顶点，e 条边的无向图，其邻接表需 n 个头结点和 2e 个表结点。显然，在边较少时，用邻接表表示图比邻接矩阵节省空间。在无向图的邻接表中，顶点 v_i 的度恰为第 i 个链表中结点的数目。在有向图的邻接表中，第 i 个链表中的结点数目为顶点 v_i 的出度；为了求入度，必须遍历整个邻接表，在所有弧结点中邻接点域的值为 i 的弧结点的个数即为 v_i 的入度。当然，有时为了便于确定顶点的入度或以顶点 v_i 为头的弧，可以建立一个有向图的逆邻接表，即对每个顶点 v_i 建立一个以 v_i 为弧头的弧结点的链表。例如，图 8-6(e)所示为有向图 G_1 的逆邻接表。

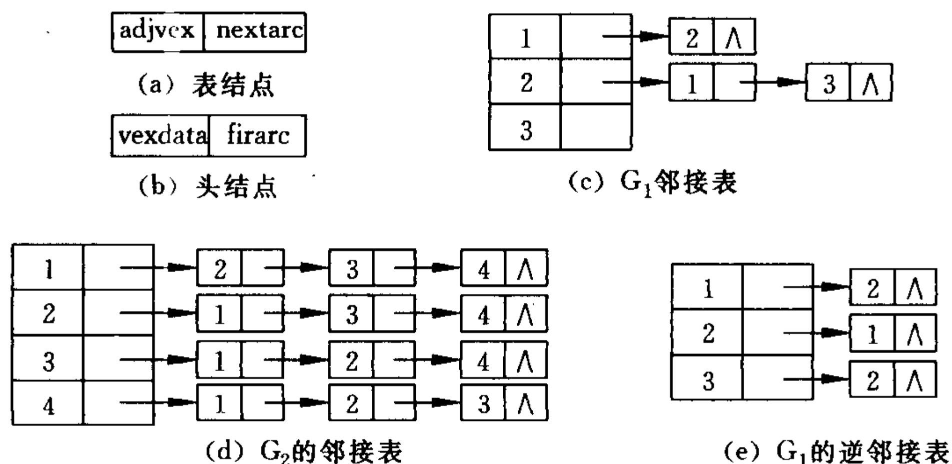


图 8-6　图的邻接表

在邻接表上容易找到任一顶点的第一个邻接顶点和下一个邻接顶点，但要判定任意两个顶点 v_i 和 v_j 之间是否有边或弧相连，则需搜索第 i 个或(和)第 j 个链表，因此，不及邻接矩阵方便。

三、十字链表

十字链表是有向图的另一种链式存储结构。可以将它看成是由有向图的邻接表和逆邻接表合起来的一种链表。在十字链表中，有向图中的每一条弧对应有一个结点表示，称为弧结点；图中的每一个顶点对应也有一个结点表示，称为顶点结点。这些结点的结构如图 8-7(a)、(b)所示。在弧结点中有四个域：其中，尾域(tailvex)和头域(headvex)分别表示弧尾和弧头两个顶点在图中的编号；链域 hlink 指向具有相同弧头的下一个弧结点，链域 tlink 指向具有相同弧尾的下一个弧结点。这样，弧头相同的弧结点通过 hlink 域链在同一链表上；弧尾相同的弧结点通过 tlink 域也链在同一链表上。顶点结点由三个域组成：其中数据域 data 用于存放顶点有关的信息，链域 firstin 和 firstout 分别指向以该顶点为弧头或弧尾的第一条弧的结点；同时，我们把所有的顶点结点用一个向量存放。例如，图 8-7(c)为图 8-1(a)所示的有向图 G_1 的十字链表存储结构。

在有向图的十字链表上，对任一顶点 v_i 求以其为弧尾的弧和求以其为弧头的弧是同样方便的。

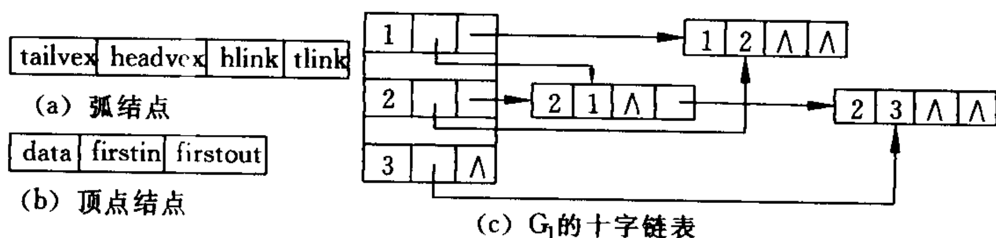


图 8-7 有向图的十字链表

四、邻接多重表

邻接多重表是无向图的另一种链式存储结构。

虽然邻接表是无向图的一种很有效的存储结构，然而由于图中的每一条边(v_i,v_j)在邻接表都有两个结点表示：一个结点在 v_j 的链表中，一个结点在 v_i 的链表中，这给图的某些操作带来了一些不便。例如，要修改一条边的有关信息或删除一条边等，都必须同时对这条边的两个结点进行操作。此时，用邻接多重表存储无向图会更适宜。

在无向图的邻接多重表中，每一条边用一个结点表示，称为边结点，其结构如图 8-8(a)所示。其中 ivex、jvex 分别表示该边依附的两个顶点在图中的编号；链域 ilink 和 jlink 分别用于指向依附于顶点 ivex 和 jvex 的下一条边的结点。图中的每一个顶点也用一个结点表示，称为顶点结点，其结构如图 8-8(b)所示。其中数据域 data 用于存储有关该顶点的信息，链域 firstedge 指向第一条依附于该顶点的边的结点。这些顶点结点可以用一个向量存储。例如，图 8-8(c)所示为图 8-1(b)中 G_2 的邻接多重表。

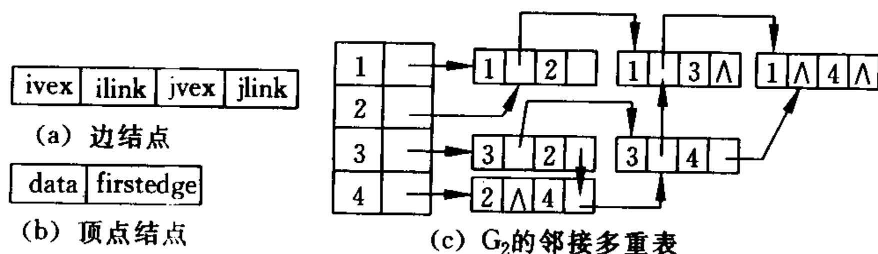


图 8-8 无向图的邻接多重表

显然，在邻接多重表中依附于同一顶点 v_i 的边通过链域 ilink 链接在同一链表中；由于每条边依附于二个顶点，故每个边结点同时处

在两个链表中。

8.3 图的遍历和连通分量

从图中某一顶点出发访问图中所有顶点，且使每一个顶点仅被访问一次，这一过程就叫做图的遍历。图的遍历类似于树的遍历，但要复杂得多。因为图中的一个顶点可能和其余任何顶点相邻接，所以在访问了某个顶点之后，沿着某条路径搜索，可能又会回到该顶点上。例如，图 8-1(b)中的 G_2，由于图中存在回路，所以当从 v_1 出发，在访问了 v_4，v_2 之后，沿着边(v_2,v_1)又回到了 v_1。为此，在遍历图时必须对已被访问过的顶点进行标志，以免重复访问。

通常有两种遍历图的方法：一种称为深度优先搜索方法，另一种称为广度优先搜索方法。

一、深度优先搜索法

深度优先搜索类似于树的先根遍历，其做法如下：从图中某一个顶点 v_0 出发，

(1)访问此顶点；

(2)从 v_0 的一个未被访问过的邻接顶点出发，深度优先搜索图。

显然，这是一个递归过程。下面我们以邻接表作为图的存储结构，具体讨论深度优先搜索算法。在此我们设一个辅助数组 visited[1..n]用于标识一个顶点是否被访问过。初始时，其每个分量 visited[i]都为假(false)，表示顶点 v_i 还没有被访问；一旦 v_i 被访问，则 visited[i]置为真(true)。图 8-9 为深度优先搜索算法的框图描述。

下面我们用 PASCAL 语言描述此算法。在这里，假设对 visited 的初始化已在主程序中完成。

```
CONST
  nmax={图中顶点的最大数目}；
TYPE
  arcptr=↑arcnode;
```

```
    arcnode=RECORD
              adjvex:integer;
              nextarc:arcptr
             END;
    vnode=RECORD
            vexdata:char;{顶点的名}
            firstarc:arcptr
           END;
    adjlist=ARRAY [1..nmax] OF vnode;
  PROCEDURE dfs( g:adjlist; v:integer; VAR visited:ARRAY
[1..nmax] of boolean);
    VAR p:arcptr;
    BEGIN
      write (g[v].vexdata); {访问 v}
      visited[v]:=true;
      p:=g[v].firstarc;
      WHILE p<>nil DO
        BEGIN
          IF NOT visited[p↑.adjvex]
            THEN dfs(g,p↑.adjvex,visited);
            p:=p↑.nextarc
        END
    END;
```

设图G有n个顶点,e条边,由于在搜索过程中每条边的结点都要被检测一次,而边结点有2e个,所以完成搜索的时间复杂度可记为O(e)。

对于如图8-10(a)所示的无向图,其邻接表如图8-10(b)所示,则按上述算法,其深度优先搜索所访问的顶点序列为A、B、D、H、E、F、C、G。

二、广度优先搜索

广度优先搜索类似于树的按层次遍历,其做法如下:从图中某一

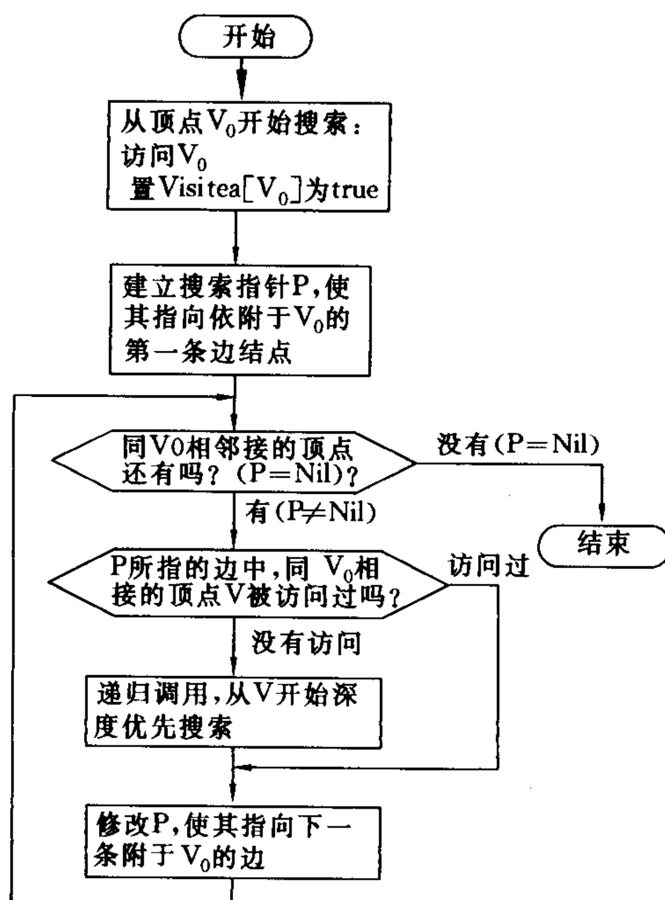


图 8-9　深度优先搜索算法的框图

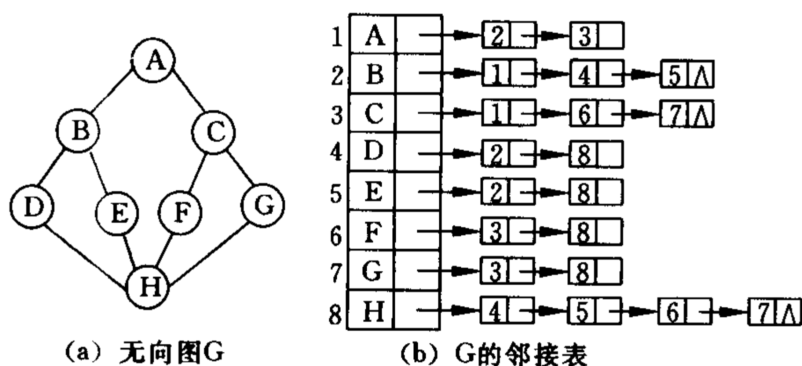


图 8-10　无向图 G 及邻接表

顶点 v_i 出发，

(1)首先访问 v_i，然后依次访问和 v_i 相邻接的顶点 v_{i1}、v_{i2}、…、v_{ik}；

(2)再按 v_{i1}、v_{i2}、…、v_{ik}的顺序，访问其中每个顶点的、所有未被访问过的邻接顶点。如此反复，直到图中所有与 v_i 相连通的顶点都

被访问到。例如，对于图 8-10(a)所示的图 G，按广度优先搜索方法，从 A 出发的顶点访问序列为 A、B、C、D、E、F、G、H。

为了实现此算法，我们需要设置一个队列。一开始把 v_i 入队，然后从队列中取顶点访问；访问完后，把同该顶点相邻接的未被访问过的顶点入队，然后再从队列中取顶点访问；…。如此重复，直到队列为空。图 8-11 为该算法的框图描述。

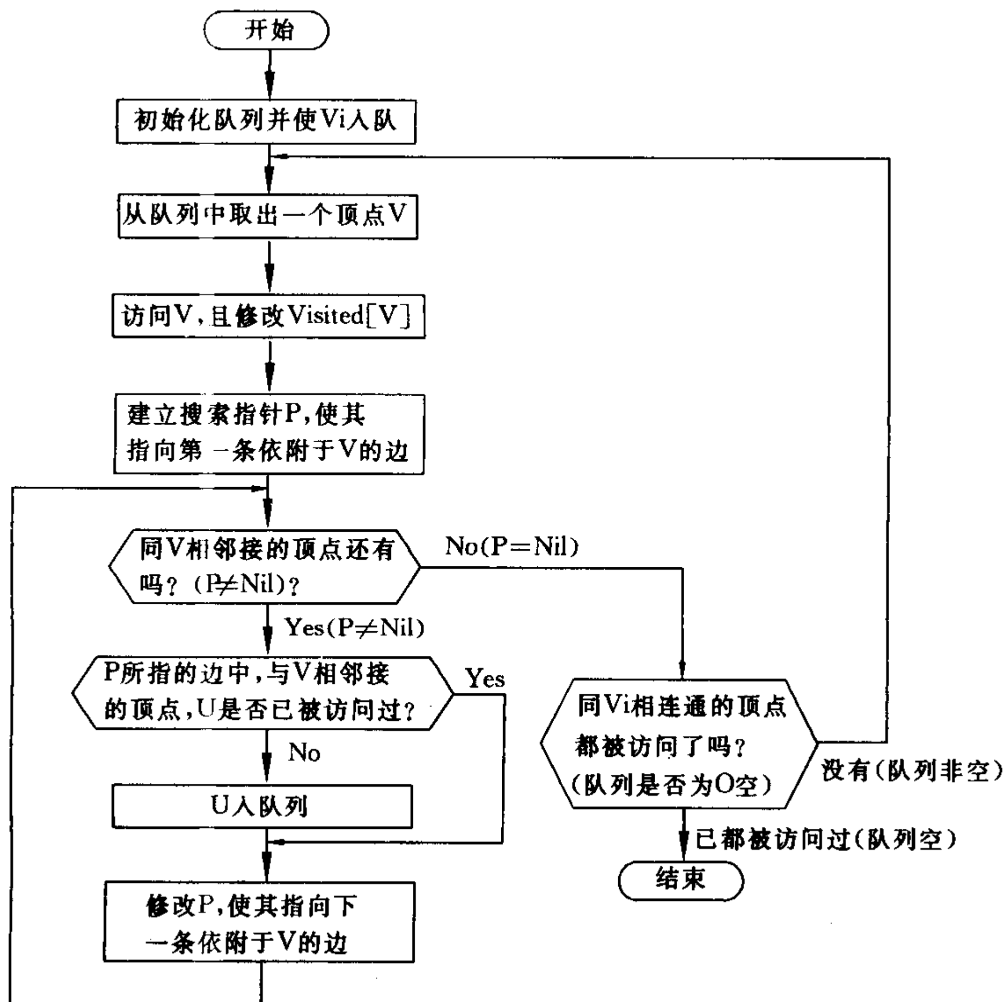


图 8-11　广度优先搜索算法的框图

该算法的 PASCAL 语言描述，请读者根据图 8-11 所示框图自已完成。该算法的时间复杂度也是 O(e)。

三、图的连通分量

在对无向图进行遍历时，对于连通图只需一次调用搜索过程(深度或广度搜索)。也就是说，从图中任一顶点出发便可访问到图中各个顶点。然而，对于非连通图，从图中一个顶点v出发，只能访问到v所在的连通分量。显然，若从无向图的每个连通分量中的一个顶点出发搜索图，就可以求出无向图的所有连通分量。对具有n个顶点的无向图，图8-12是求其连通分量算法的框图描述。

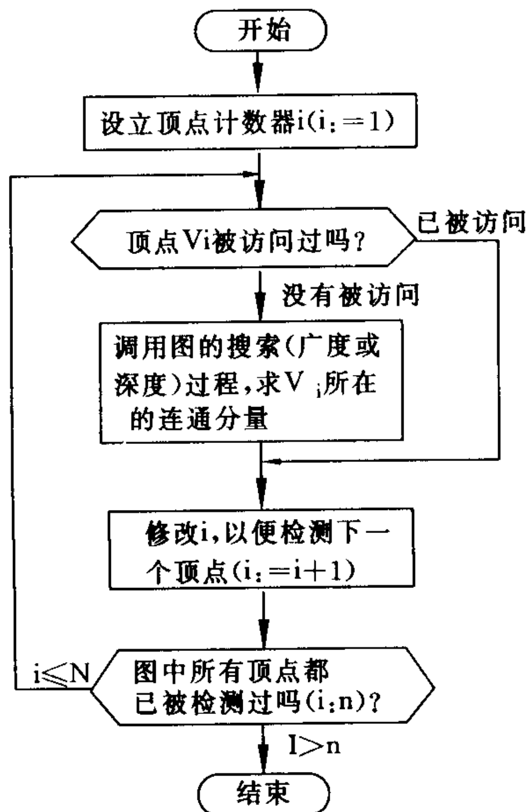


图 8-12　求无向图连通分量的算法框图

8.4　最小生成树

前面，我们介绍了生成树的概念。显然，一个连通图的生成树不一定是唯一的。例如，对图8-10(a)所示的无向图G，当分别按深度和广度优先搜索法进行遍历时就可以得到图8-13(a)、(b)所示的两棵不同的生成树，并分别称为深度优先生成树和广度优先生成树。

利用生成树可以解决一些实际工程问题。例如，要在n个城市之间建立通讯联络网，而连通n个城市至少需要n－1条通讯线路。若把城市看成图的顶点，通讯线路看成边，则该图的任一生成树就是一个可行的建造通讯网的方案。由于n个城市之间，可行线路有n(n－1)/2条，那么，如何选择建造代价最小的、连通n个城市的n－1条

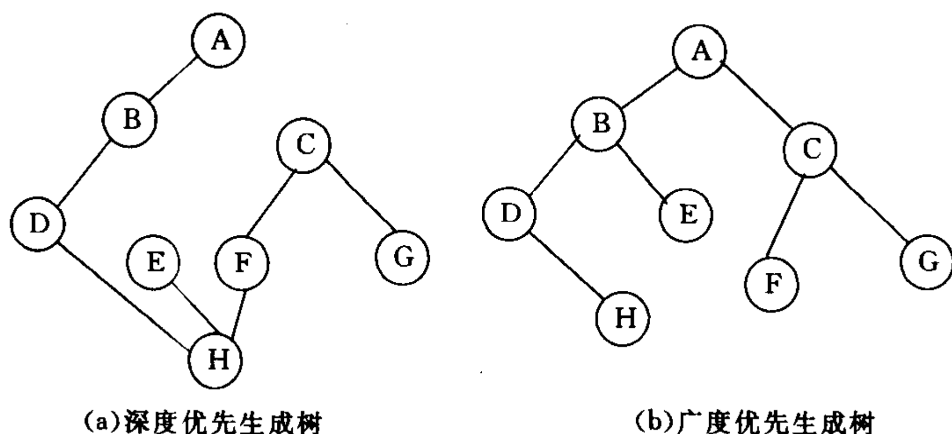


图 8-13　图 G 的两棵生成树

线路呢？这就是我们接下来要讨论的构造最小生成树的问题。

我们把边上赋以权值的图称为网或带权图。所谓最小生成树就是该生成树中所有边上的权值之和达到最小。构造最小生成树的算法很多，它们一般都是按如下的原则进行：尽可能的选取权值小的边，但不能构成回路；直到在网中选择了 n－1 条边以连通网的 n 个顶点。下面我们介绍两个典型的构造最小生成树的算法：普里姆(Prim)算法和克鲁斯卡尔(Kruskal)算法。

一、普里姆算法

设 N＝(V，E)是连通网，TE 是 N 上最小生成树中边的集合，U 是 V 的一个非空子集。任选一个顶点 u_0，算法从 $U=\{u_0\}(u_0\in V)$，TE＝Φ 开始，重复执行下述操作：在所有 u∈U，v ∈V－U 的边(u，v)中找到一条代价最小的边(u′，v′)并入集合 TE，同时把 v′并入 U，直到 U＝V，或 TE 中有 n－1 条边为止，则所得的 T＝(V，TE)为 N 上的最小生成树。

为了实现这个算法需附设一个辅助数组 closedge[1..n]，对于每个不在 U 中的顶点 v(v∈V－U)，对应有一个数组分量 closedge[v]，用于记录依附于顶点 v 和集合 U 中顶点的具有最小权值的那条边(u，v) (u∈U)。每个数组分量 closedge[v]有两个域，其中 closedge[v]. mincost 用于存放那条边的权值，closedge[v]. vex 用于存放顶点 u 的编号。图 8-14(b)～(f)展示了对图 8-14(a)所示的连通

网按普里姆算法构造最小生成树的过程。在构造过程中辅助数组 closedge 的变化过程，如图 8-15 所示。其中 closedge[v]. mincost＝0 表示 v∈U，closedge[v]＝∞表示 v 同 U 中顶点不邻接。开始时 U＝{1}，所以在所有 u∈U，v∈V－U 的边(u，v)中找到的权值最小的边(u′，v′)就是(1，3)，它就是生成树上的一条边，同时将顶点 v′(v′＝3)并入集合 U(即使 closedge[3]. mincost＝0)；然后，对所有 V－U 中的顶点 v，查看是否要修改 closedge[v]，即当边(3，v)上的权值小于 closedge[v]. mincost 时(说明 v 和集合 U 中顶点有更小代价的边)就要进行修改。在例子中修改了 closedge[5]和 closedge[4]。如此重复，直到 U＝V。

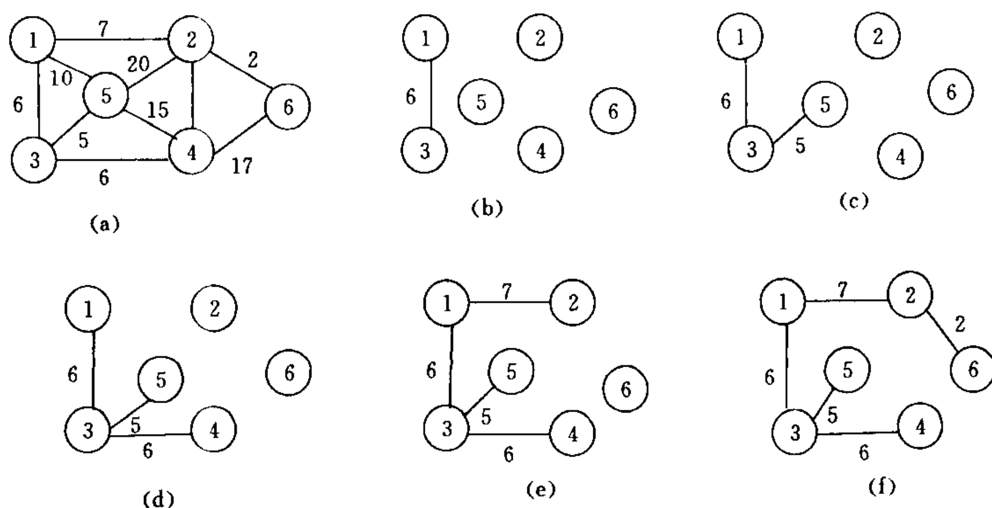


图 8-14　普里姆算法构造最小生成树的过程

closedge \ V	1	2	3	4	5	6	(u_0,v_0)	U	V-U
vex		1	1		1		(1,3)	{1}	{2,3,4,5,6}
mincost	0	7	6	∞	10	∞			
vex		1	1	3	3		(3,5)	{1,3}	{2,4,5,6}
mincost	0	7	0	6	5	∞			
vex		1	1	3	3		(3,4)	{1,3,5}	{2,4,6}
mincost	0	7	0	6	0	∞			
vex		1	1	3	3	A	(1,7)	{1,3,4,5}	{2,6}
mincost	0	7	0	0	0	17			
vex		1	1	3	3	2	(2,6)	{1,3,4,5,2}	{6}
mincost	0	0	0	0	0	2			
vex		1	1	3	3	2		{{1,3,4,5,2,6}	φ
mincost	0	0	0	0	0	0			

图 8-15　最小生成树构造过程中辅助数组的变化过程

在下面具体讨论普里姆算法实现时，我们假设网 N 以耗费(邻接) 矩阵作为存储结构，其定义如下：

$$cost[i,j]=\begin{cases} w_{ij} & (v_i,v_j)\in E; \\ \infty & 否则 \end{cases}$$

其中，w_{ij}表示边(v_i,v_j)上的权值，∞表示计算机允许的最大整数。

图 8-16 是普里姆算法的框图描述：

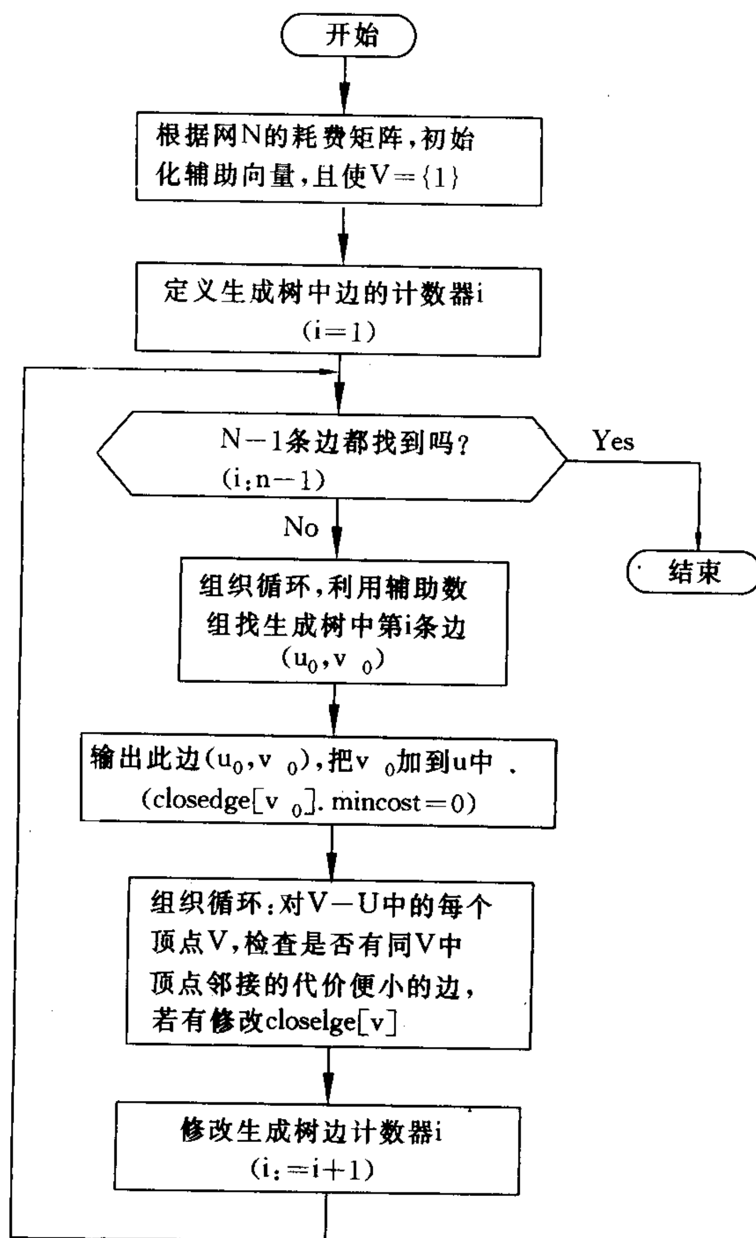


图 8-16　普里姆算法框图

该算法的 PASCAL 语言描述如下：

```
        PROCEDURE minspamtree-prim (gn: ARRAY [1.. n, 1.. n] OF
integer);
      VAR v,v0,w,i:integer;
      BEGIN
        FOR v:=2 TO n DO
        BEGIN
          closedge[v].vex:=1; closedge[v].mincost:=gn[1,v]
        END;
        closedge[1].mincost:=0;
        FOR i:=1 TO n-1 DO
          BEGIN
            w:=maxint;
            FOR v:=2 TO n DO
            IF (closedge[v].mincost<>0) AND (closedge[v].mincost<w)
              THEN BEGIN
                     w:=closedge[v].mincost;
                     v0:=v
                   END;
            write(closedge[v0].vex,v0);
            closedge[v0].mincost:=0;
            FOR v:=2 TO n DO
              IF gn[v0,v]<closedge[v].mincost
                THEN BEGIN
                       closedge[v].mincost:=gn[v0,v];
                       closedge[v].vex:=v0
                     END
          END
        END;
```

普里姆算法的时间复杂度为 $O(n^2)$，同网中的边数无关，所以适合于求稠密网的最小生成树。

二、克鲁斯卡尔算法

对于连通网 N=(V,E),开始我们假设网中每一个顶点自成一个连通分量。然后在 E 中选择一条权值最小的边(v_i,v_j),若顶点 v_i、v_j 分别属于两个不同的连通分量,则加入此边把这两个连通分量合并为一;否则,舍去此边,再选择下一条权值最小的边。如此重复,直到选出 n－1 条边(即合并为一个连通分量)为止。

图 8-17(a)～(e)展示了对图 8-14(a)所示的连通网按克鲁斯卡尔算法构造最小生成树的过程。

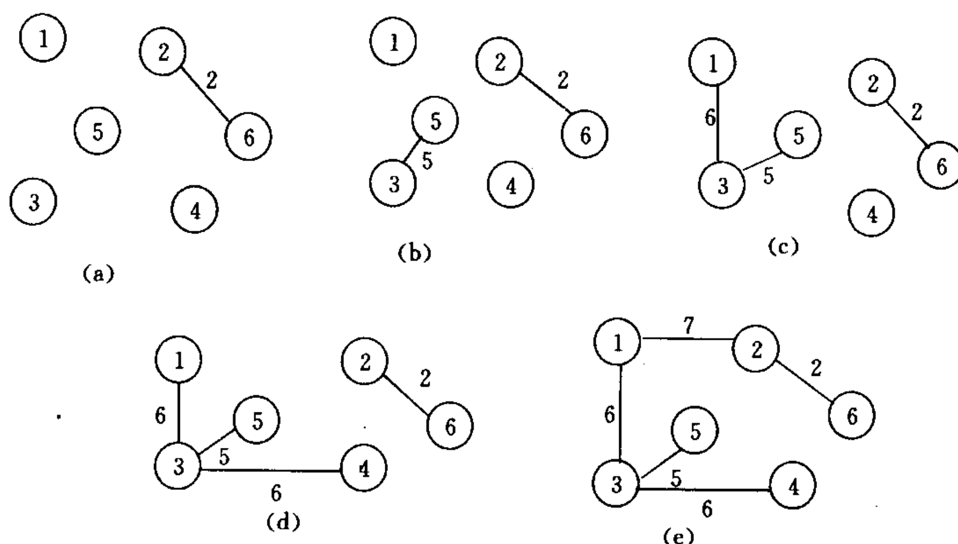


图 8-17 克鲁斯卡尔算法构造最小生成树的过程

克鲁斯卡尔算法的时间复杂度与网中边的数目 e 有关,为 O(eloge),它适用于求稀疏网的最小生成树。

8.5 最短路径

假如要在计算机上建立一个交通咨询系统,则可以采用图的结构来表示实际的交通网络。其中,用图的顶点表示城市,用图的边表示城市间的公路,并考虑把城市间的距离或公路上的费用作为边上的权值。这个咨询系统可以回答旅客各种各样的问题。例如:从 A 城

到 B 城是否有通路？有多少条通路？那一条所需时间最少？那一条途中经过的城市最少？等等。这就是我们本节中要讨论的最短路径问题。这里，所谓最短路径是指路径上所有边的权值之和为最小的路径，而不是指路径上经过的边的数目最少的路径。考虑到交通道路的有向性，所以下面的讨论将针对有向带权图进行，并称路径的开始顶点为源点，最后一个顶点为终点。另外，设弧上的权值都为正值。

本节给出两个算法：一个是求从指定源点到其余各顶点的最短路径；另一个是求任意一对顶点间的最短路径。

一、从某一源点到其余各顶点的最短路径

在这里，我们要讨论：对于给定带权有向图 G 和源点 v，求从 v 到 G 中其余各顶点的最短路径。对于这个问题，一种直观的方法是罗列从 v 到其余一顶点的所有可能路径，并进行比较，选择其中最短的。例如，对于图 8-18 所示的带权有向图，设源点为 v_1，则从 v_1 到 v_2 有三条路径：其一是 (v_1,v_2)，长度为 50；其二是 (v_1,v_3,v_4,v_2)，长度为 45；其三是 (v_1,v_5,v_4,v_2)，长度为 95。经比较，可知从 v_1 到 v_2 的最短路径为 (v_1,v_3,v_4,v_2)。依此类推，可求出 v_1 到其余各顶点的最短路径，共 n－1 条。这种方法虽然简单、直观，然而效率却不高，且难于在计算机上实现。

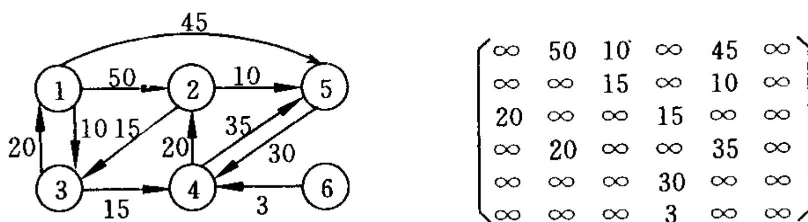


图 8-18　带权有向图及其耗费矩阵

那么如何在计算机上方便地求解这个问题呢？迪杰斯特拉(Dijkstra)提出了一个按路径长度递增的次序产生最短路径的算法。该算法在寻找最短路径过程中，把带权有向图中的顶点分成两个集合 S 和 T。凡以 v_1 为源点已求出最短路径的终点属于集合 S，初始时 S 只包含 v_1；集合 T 包含所有尚未确定最短路径的顶点，初始时 T 包

含带权有向图中除 v_1 以外的其它所有顶点。算法按各顶点与 v_1 间的最短路径长度递增的次序将集合 T 中的顶点加入到集合 S 中。显然，在这一过程中，v_1 到集合 S 中各顶点的最短路径长度始终不大于 v_1 到集合 T 中各顶点的最短路径长度。为了能方便地实现这个算法，我们引进一个辅助向量 dist[1..n]，它的每个分量 dist[i]表示在寻找过程中当前所找到的从 v_1 到每个终点 v_i 的最短路径(不一定是真正所要找的最短路径)长度。它的初态是：若从 v_1 到 v_i 有弧，则 dist[i]为弧上的权值；否则，dist[i]为∞。显然，从 v_1 到其余各顶点的最短路径中最短的一条路径长度为：

$$dist[k]=\min\{dist[i] \mid v_i \in T\}$$

此路径为(v_1,v_k)。这时把编号为 k 的顶点从 T 中删除，加入到 S 中。例如，对图 8-18，从 v_1 出发第一次选中的最短路径为(v_1,v_3)。那么下一条长度次短的最短路径是哪一条呢？首先，对于集合 T 中的每一顶点 v_j 来说，当把 v_k 加入到 S 中之后，其最短路径或者仍为(v_1,v_j)或者为(v_1,v_k,v_j)，而不会有其它选择。也就是说，此时从 v_1 到 v_j 的最短路径或者仍是原来的，或者是通过 v_1 到 v_k 的已确定的最短路径后，再从 v_k 到 v_j。所以，对集合 T 中每个顶点 v_j，当 dist[k]+cost[k,j]<dist[j]时，修改 dist[j]，使其为：dist[j]:=dist[k]+cost[k,j]。例如，对图 8-18 来说，当把顶点 v_3 加入到 S 中之后，原来 dist[4]=∞，要改为 dist[4]=25。接下来，我们就可以通过向量 dist[1..n]，寻找长度次短的最短路径。例如，对图 8-18 来说，长度次短路径为从 v_1 到 v_4 的最短路径(v_1,v_3,v_4)。如此重复，我们就可以求出从 v_1 到其余各顶点的最短路径。

根据上述分析，我们可以用图 8-19 所示的框图来描述迪杰斯特拉算法。

根据此框图，我们可用 PASCAL 语言描述此算法。

```
TYPE
  st=set of 1..n;
PROCEDURE shortpath-dij(cost:ARRAY [1..n,1..n] of integer;
VAR dist[1..n] OF integer; VAR path:ARRAY [1..n] OF st)
```

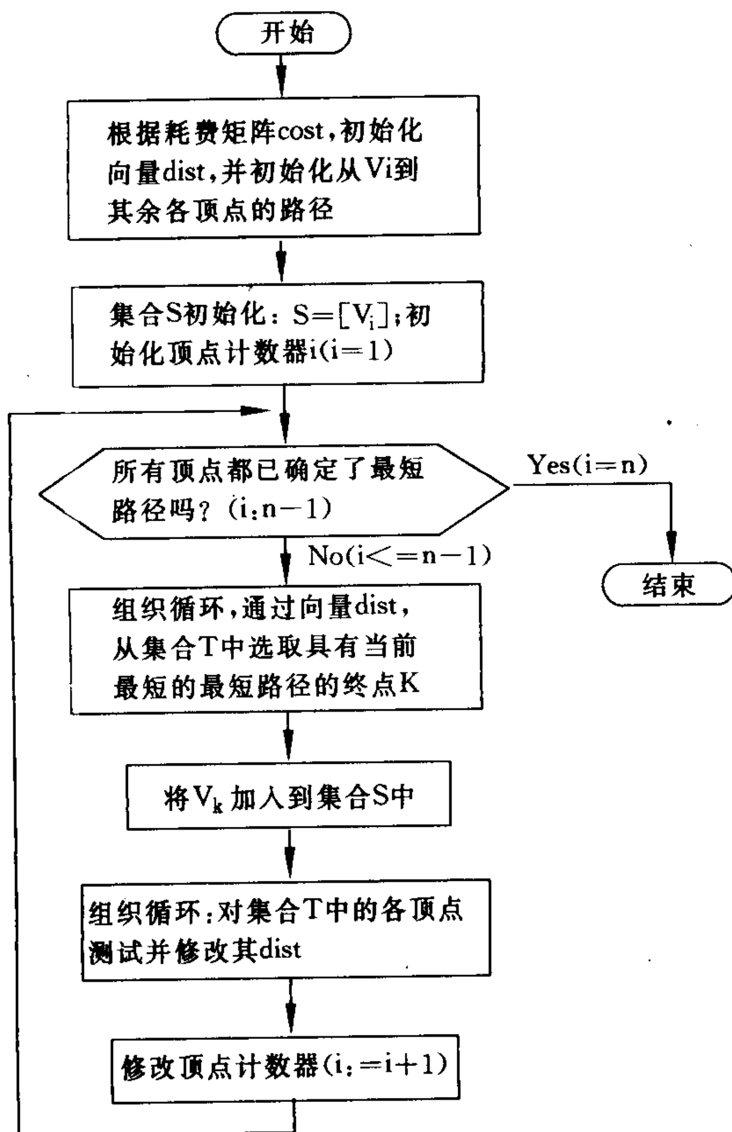


图 8-19　迪杰斯特拉算法框图

```
VAR s:st;
      i,k,w:integer;
BEGIN
  FOR i:=1 TO n DO
    BEGIN
      dist[i]:=cost[1,i];
      IF dist[i]<maxint
        THEN path[i]:=[1]+[i]
        ELSE path[i]:=[ ]
```

```
    END;
  s=[1];
  FOR i:=1 TO  n-1  DO
    BEGIN
      w:=maxint; k:=1;
      FOR j:=1 TO n DO
        IF NOT (j in s) AND (dist[j]<w)
          THEN BEGIN k:=j; w:=dist[j] END;
      s:=s+[k];
      FOR j:=1 TO n DO
        IF NOT (j in s) AND (dist[k]+cost[k,j]<dist[j])
          THEN BEGIN
                 dist[j]:=dist[k]+cost[k,j];
                 path[j]:=path[k]+[j]
               END
    END
END;
```

对于图 8-18 所示的带权有向网，采用上述算法，其计算过程中 dist 和 path 的变化情况如图 8-20 所示。

迭代次数	集合 S	所选顶点	dist[i]和 path[i]					
			1	2	3	4	5	6
初始	{1}		0*	50 (V_1,V_2)	10 (V_1,V_3)	∞	45 (V_1,V_5)	∞
1	{1,3}	3	0*	50 (V_1,V_2)	10* (V_1,V_3)	25 (V_1,V_3,V_4)	45 (V_1,V_5)	∞
2	{1,3,4}	4	0*	45 (V_1,V_3,V_4,V_2)	10* (V_1,V_3)	25* (V_1,V_3,V_4)	45 (V_1,V_5)	∞
3	{1,3,4,2}	2	0*	45* (V_1,V_3,V_4,V_2)	10* (V_1,V_3)	25* (V_1,V_3,V_4)	45 (V_1,V_5)	∞
4	{1,3,4,2,5}	5	0*	45* (V_1,V_3,V_4,V_2)	10* (V_1,V_3)	25* (V_1,V_3,V_4)	45* (V_1,V_5)	∞
5	{1,3,4,2,5}	无						

图 8-20 从 V_1 到其余各顶点的最短路径计算过程

分析这个算法，我们可看出，第一个 FOR 循环的时间为 O(n)，

第二个 FOR 循环共进行 n－1 次，每次内循环的执行时间为 $O(n)$，所以总的时间是 $O(n^2)$。

二、每一对顶点间的最短路径

现在我们要讨论带权有向图中，求任何一对顶点间的最短路径的问题。我们可以每次以一个顶点为源点，重复执行上述迪杰斯特拉算法 n 次，这样，便可以求出图中每一对顶点之间的最短路径。显然，其时间复杂度为 $O(n^3)$。

我们要介绍求解这一问题的另外一个算法。它的时间复杂度虽然仍为 $O(n^3)$，但形式上要简单一些。这个算法由弗洛伊德(Floyd)提出。算法中仍以耗费矩阵 cost 作为带权有向图的存储结构，其算法思想是：

假设求顶点 v_i 到 v_j 的最短路径。开始，我们把 cost[i,j]作为从 v_i 到 v_j 的路径上没有中间顶点的最短路径长度(当长度为∞时，表示无路径)。当然，这条路径是否为要找的最短路径，还需进行 n 次试探：首先考虑让路径经过顶点 v_1，比较 (v_i,v_j) 和 (v_i,v_1,v_j) 的路径长度，取两者中较短者为从 v_i 到 v_j 的路径上中间顶点序号不大于 1 的最短路径。在这基础上再考虑让路径经过顶点 v_2，也就是说，若我们已经有了 $(v_i,\cdots,v_j)$、$(v_i,\cdots,v_2)$、$(v_2,\cdots,v_j)$，它们分别表示从 v_i 到 v_j、从 v_i 到 v_2、从 v_2 到 v_j 的中间顶点序号不大于 1 的最短路径，那么，就可以比较 $(v_i,\cdots,v_2,\cdots,v_j)$ 和 $(v_i,\cdots,v_j)$ 的路径长度，取较短者为从 v_i 到 v_j 的路径上中间顶点序号不大于 2 的最短路径。再考虑让路径经过顶点 v_3，依次类推。一般情况下，若我们已经有了从 v_i 到 v_k、从 v_k 到 v_j 和从 v_i 到 v_j 的中间顶点序号不大于 k－1 的最短路径 $(v_i,\cdots,v_k)$、$(v_k,\cdots,v_j)$、$(v_i,\cdots,v_j)$，则我们接下来考虑让路径经过顶点 v_k，比较 $(v_i,\cdots,v_k,\cdots,v_j)$ 和 $(v_i,\cdots,v_j)$ 的路径长度，取较短者为从 v_i 到 v_j 的路径上中间顶点序号不大于 k 的最短路径。如此进行 n 次试探，就得到了从 v_i 到 v_j 的路径上中间顶点序号不大于 n 的最短路径，这也就是我们要求的最短路径。

为了实现上述算法，我们定义一个 n 阶方阵的序列：

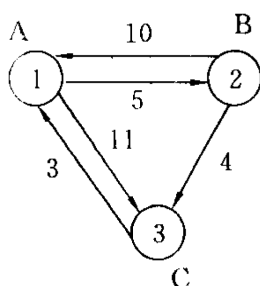
$$A^{(0)}, A^{(1)}, \cdots A^{(k)}, \cdots, A^{(n)}$$

其中，$A^{(0)}[i,j] = cost[i,j]$

$$A^{(k)}[i,j] = \min\{A^{(k-1)}[i,j], A^{(k-1)}[i,k] + A^{(k-1)}[k,j]\}$$

$$1 <= k <= n$$

由上述计算公式可见，$A^{(k)}[i,j]$表示从 v_i 到 v_j 的中间顶点序号不大于 k 的最短路径长度。另外，为了记录从 v_i 到 v_j 的最短路径，需要再定义一个 n 阶方阵的序列 $path^{(0)}, path^{(1)}, \cdots, path^{(n)}$。其中 $path^{(k)}[i,j]$用于记录从 v_i 到 v_j 的中间顶点序号不大于 k 的最短路径。例如，对图 8-21 所示的带权有向图，按照弗洛伊德算法，可得到如图 8-22 所示的两个矩阵序列。



(a)带权有向图

$$cost = \begin{bmatrix} \infty & 5 & 11 \\ 10 & \infty & 4 \\ 3 & \infty & \infty \end{bmatrix}$$

(b)耗费矩阵

图 8-21　带权有向图及其耗费矩阵

$$A^{(0)} = \begin{bmatrix} \infty & 5 & 11 \\ 10 & \infty & 4 \\ 3 & \infty & \infty \end{bmatrix} \qquad A^{(1)} = \begin{bmatrix} \infty & 5 & 11 \\ 10 & \infty & 4 \\ 3 & 8 & \infty \end{bmatrix}$$

$$A^{(2)} = \begin{bmatrix} \infty & 5 & 9 \\ 10 & \infty & 4 \\ 3 & 8 & \infty \end{bmatrix} \qquad A^{(4)} = \begin{bmatrix} \infty & 5 & 9 \\ 7 & \infty & 4 \\ 3 & 8 & \infty \end{bmatrix}$$

$$path^{(0)} = \begin{bmatrix} / & AB & AC \\ BA & / & BC \\ CA & / & / \end{bmatrix} \qquad path^{(1)} = \begin{bmatrix} / & AB & AC \\ BA & / & BC \\ CA & CAB & / \end{bmatrix}$$

$$path^{(2)} = \begin{bmatrix} / & AB & ABC \\ BA & / & BC \\ CA & CAB & / \end{bmatrix} \qquad path^{(3)} = \begin{bmatrix} / & B & ABC \\ BCA & / & BC \\ CA & CAB & / \end{bmatrix}$$

图 8-22　弗洛伊德算法计算过程中各对顶点间的最短路径及长度

图 8-23 是弗洛伊德算法的框图描述。

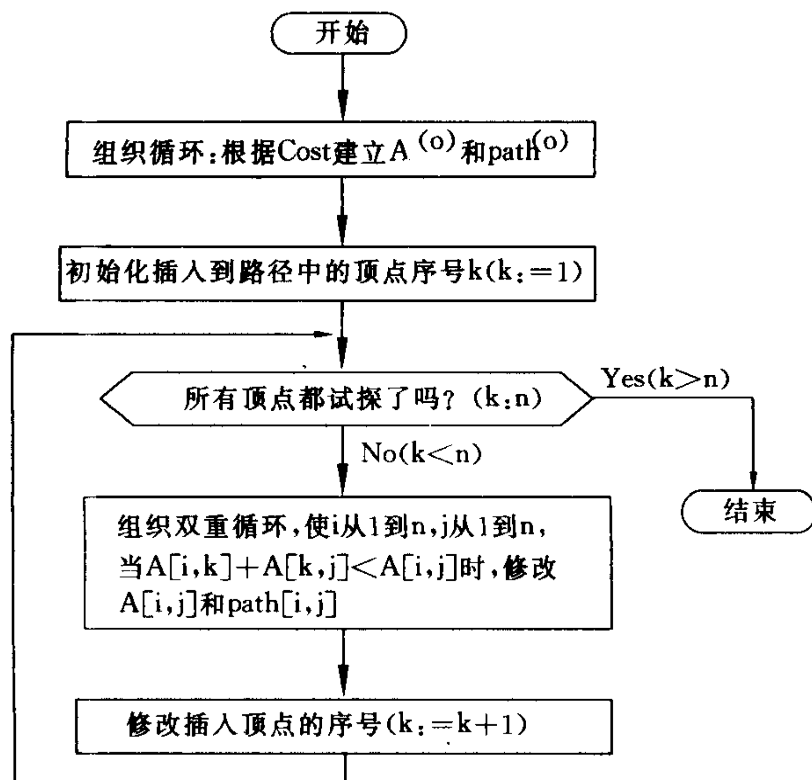


图 8-23 弗洛伊德算法框图

这个算法的 PASCAL 语言描述,请读者根据框图自已完成。

8.6 有向无环图及其应用

一、有向无环图

所谓有向无环图就是指没有环的有向图,简称 DAG 图。例如,图 8-24(a)、(b)分别为有向无环图和非 DAG 图。

有向无环图是描述一项工程进度安排或一个系统运行过程的有效工具。除了最简单的情况外,一个工程一般均可分为若干个被称为活动的子工程,而这些子工程之间通常受着一定条件的约束,如其中某些子工程的开始,必须在另一些子工程完成之后等。用 DAG 图描述工程进度安排,一般有二种形式:其一是以图中的顶点表示活动,弧表示活动之间的先后关系,这时有向无环图称为用顶点表示活动

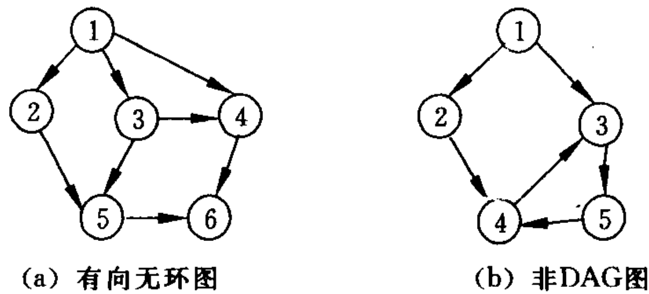


图 8-24 有向无环图与非 DAG 图

的网(Activity On Vertex network),简称为 AOV 网;其二是以图中的弧表示活动,以顶点表示活动的开始或结束等事件,并且以弧上的权值表示活动所持续的时间,这时有向无环图称为用边表示活动的网(Activity On Edge network),简称为 AOE 网。对于一个工程或一个生产流程,人们一般关心的最基本的两个问题是:第一,工程能否顺利进行?第二,要估算整个工程完成所必须的最短时间。下面分别讨论这两个问题。

二、拓扑排序

所谓拓扑排序就是由某个集合上的一个偏序得到该集合上的一个完全序。也就是说要构造 AOV-网中所有顶点的线性序列,使得此序列中不仅保持网中各顶点间原有的先后关系,而且使原来没有先后关系的顶点之间建立一种人为的先后关系,这种线性序列称为拓扑有序序列。构造 AOV-网的拓扑有序序列的过程即为拓扑排序。直观地看,偏序集合中只有部分元素可比较,完全序集合中全体元素都可比较。例如,图 8-25 所示,(a)表示偏序,其中,V2 与 v3 之间无法比较;(b)表示完全序。

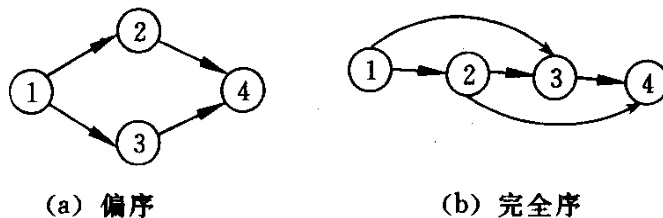
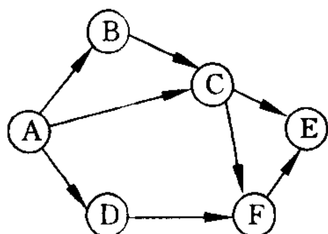


图 8-25 表示偏序和完全序的有向图

如果一个 AOV-网的所有顶点都在它的拓扑有序序列中，则该 AOV-网中不存在有向回路，该网所表示的工程按拓扑有序序列中的先后次序安排其子工程，工程就可顺利完成。当然，一个 AOV-网，其拓扑有序序列不一定唯一。例如，图 8-26(a)所示的 AOV-网，其拓扑有序序列有以下三个(A,B,C,D,F,E)、(A,B,D,C,F,E)和(A,D,B,C,E,F)。



(a) AOV-网

	顶点	入度	边表
1	A	0	→ 2 → 3 → 4 ∧
2	B	1	→ 3 ∧
3	C	2	→ 5 → 6 ∧
4	D	1	→ 6 ∧
5	E	2	∧
6	F	2	→ 5 ∧

(b) 邻接表

图 8-26　AOV-网及其邻接表

如何进行拓扑排序呢？我们可以按如下步骤进行：

(1) 在 AOV-网中选择一个没有前驱的顶点 v_i，并输出之；

(2) 在 AOV-网中删除顶点 v_i 和以 v_i 为弧尾的弧；

(3) 重复(1)和(2)，直到网中所有顶点都被输出，或网中不存在没有前驱的顶点。前者说明网中没有环，后者说明网中有环。

以图 8-26(a)中的有向图为例，首先选取没有前驱的顶点 A 进行输出，并在网中删去顶点 A 和从 A 发出的弧〈A,B〉、〈A,D〉。然后，再选取没有前驱的顶点，这时，顶点 B 和 D 都没有前驱，可以任选一个，比如 D，输出之，并删除 D 及〈D,E〉。依次类推，最后将网中的顶点全部输出，就得到了该网的一个拓扑有序序列 (A,D,B,C,F,E)。

那么，如何在计算机上实现上述算法呢？首先，我们要解决 AOV-网的存储结构问题。根据算法进行的操作，可以选择邻接表作为 AOV-网的存储结构，且在头结点中增加一个存放顶点入度的数据域(indegree)。入度为零的顶点就是没有前驱的顶点。例如，图 8-26(a)所示的有向图的邻接表如图 8-26(b)所示。当要删除从某顶点发出的弧时，可以对该弧所到达的顶点的入度减 1 来实现。同时，

为了避免重复检测入度为零的顶点，可以设置一个堆栈用于存放入度为零的顶点。这个堆栈可以另外开辟，也可以借用头结点中值为零的入度域来存放堆栈中的指针（以指示下一个入度为零的顶点序号）。以图 8-26(a)所示的有向图为例，图 8-27(b)为拓扑排序前，入度为零的顶点入栈后的状态。其中，top 为栈顶指针；图 8-27(c)表示顶点 A 输出之后，由于 B,D 的入度相继为零而入栈，则栈顶指针指向 D,D 的指针指向 B,B 的指针为零(表示栈底)。依次类推。

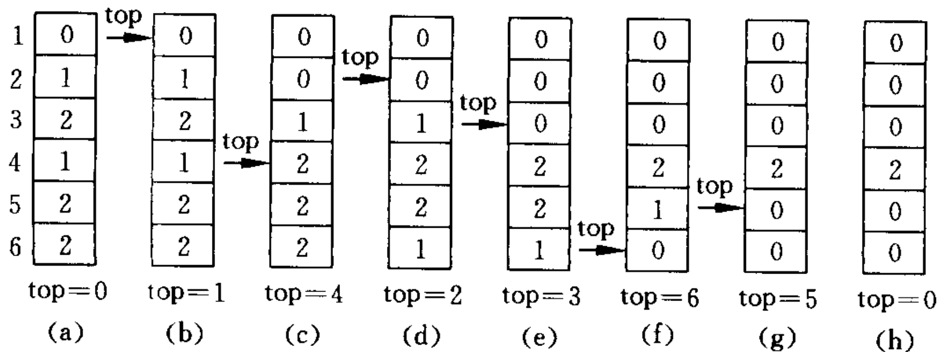


图 8-27 拓扑排序过程中入度域的变化情况

图 8-28 为上述算法的框图描述。

其 PASCAL 语言描述如下：

```
TYPE vexnode=RECORD
              vexdata  :char;
              indegree:integer;
              firstarc:arcptr;
             END;
    adjlisttp:=ARRAY [1..nmax] OF vexnode;
FUCTION topsort(VAR dig:adjlisttp; n:integer):boolean;
  VAR p:arcptr;
      i,m,top :integer;
  BEGIN
    top:=0;
    FOR i:=1 TO n DO
      IF dig[i].indegree=0 THEN
        BEGIN
          dig[i].indegree:=top;
          top:=i
```

```
        END;
    m:=0;
    WHILE top<>0 DO
      BEGIN
        write(dig[top].vexdata); m:=m+1;
        p:=dig[top].firstarc;
        top:=dig[top].indegree;
        WHILE p<>nil DO
          BEGIN
            k:=p↑.adjvex;
            dig[k].indegree:=dig[k].indegree-1;
            IF dig[k].indegree=0 THEN
              BEGIN
                dig[k].indegree:=top;
                top:=k
              END;
            p:=p↑.nextarc
          END
      END;
    IF m<n THEN return(false)
            ELSE return(true)
  END;
```

对于一个有 n 个顶点和 e 条弧的有向图，算法搜索入度为 0 的顶点的时间为 O(n)；当网中不存在环时，每个顶点需入栈一次和出栈一次，而且顶点入度减 1 的操作需执行 e 次；所以，总的时间复杂度为 O(n+e)。

三、关键路径

上面我们运用 AOV-网讨论了一个工程能否顺序进行的问题，下面我们运用 AOE-网讨论：完成一个工程至少需多少时间？以及哪些活动是影响工程进度的关键？

用 AOE-网表示一项工程的施工计划时，顶点所表示的事件是指所有以该顶点为弧头的弧所表示的活动都已完成，以及以该顶点为弧尾的弧所表示的活动都可以开始。例如，图 8-29(a)所示是一个

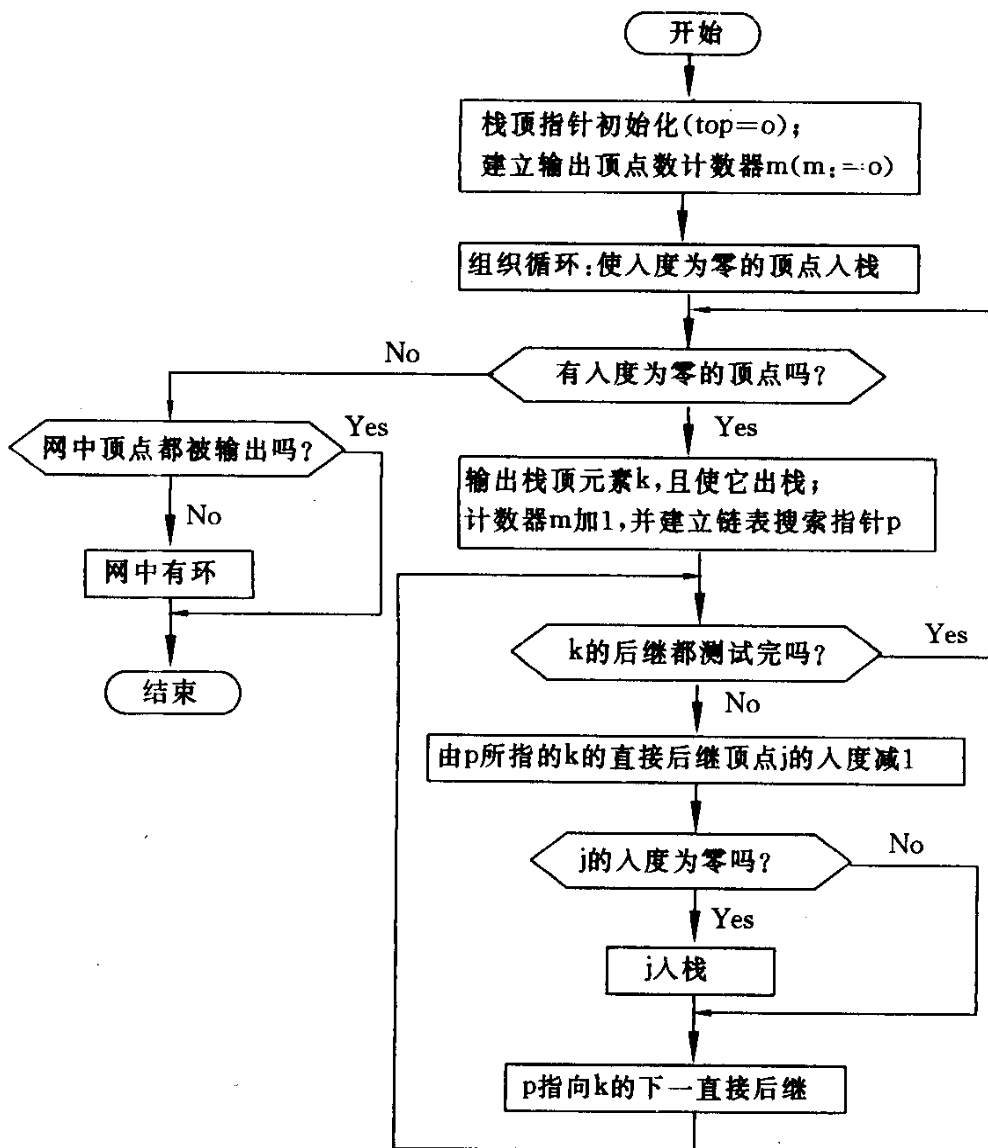


图 8-28 拓扑排序的算法框图

具有 8 个子工程(活动)的 AOE-网,网中有六个顶点,分别表示六个事件。其中,顶点 v_1 表示整个工程可以开始,即子工程 a_1、a_2 可以开工,顶点 v_4 表示子工程 a_2、a_5 完工而子工程 a_7 可以开工,等等。弧上的权值表示该弧所代表的活动的计划用时间。例如,活动 a_1 计划需花 3 天时间完成。

由于整个工程只有一个开始点和一个完成点,分别表示整个工程的开工和完工,所以,在正常情况下,AOE-网中不仅无环,而且只有一个入度为零的顶点和一个出度为零的顶点,分别称为源点和

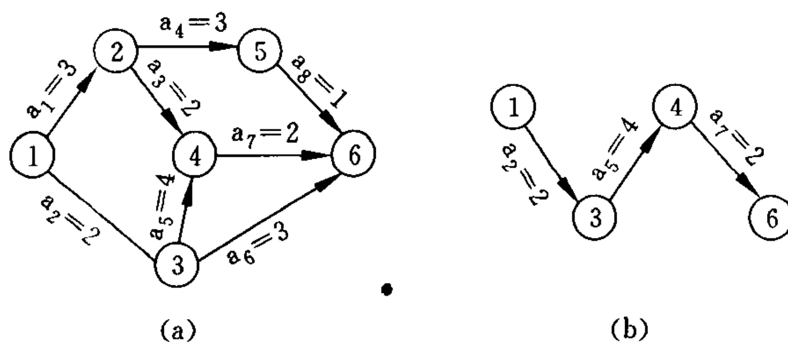


图 8-29　AOE-网及其关键路径

汇点。

在 AOE-网中有些活动能同时进行，所以，完成整个工程所必需的时间是从开始顶点到结束顶点的最长的带权路径的长度，这条路径称为关键路径。关键路径上的所有活动称为关键活动。显然，任何一项关键活动若不能按期完成，都要影响到整个工程的工期；而提高关键活动的速度，可以缩短工期。例如，在图 8-29 所示的 AOE-网中，关键路径为(v_1,v_3,v_4,v_5)，其长度为 $2+4+2=8$，关键活动为 a_2、a_5、a_7。如果提高 a_5 的速度，使其由原计划的 4 天减少到用 3 天完成，则整个工程可提前一天完工。但若进一步提高 a_5 的速度，使其只用 2 天完成，整个工程是否可以在 6 天内完成呢？显然不能，因为当 $a_5=2$ 时，关键路径就变化了。如果在 AOE-网中存在两条以上的关键路径，必需同时提高这几条关键路径上某个关键活动的速度，才能缩短工期。而提高非关键活动的速度不能加快整个工程的进度。

由以上分析可知，要对工程计划进行有效的安排和调度，首先应求出其对应的 AOE-网的关键路径和关键活动。为了计算关键活动，先定义几个有关的变量。

1. 事件的最早发生时间 Ve(j)

Ve(j)是指从源点 v_1 到顶点 v_j 的最长带权路径长度，这个时间决定了所有从 v_j 发出的弧所表示的活动能够开工的最早时间。

Ve(j)的计算方法如下：

$Ve(1)=0$；

$Ve(j)=\max\{Ve(i)+dut(\langle v_i,v_j\rangle)\}$，$\langle v_i,v_j\rangle\in T$、$2<=j<=n$。

其中，T 表示所有以顶点 v_j 为弧头的弧，$dut(\langle v_i,v_j\rangle)$ 为弧 $\langle v_i,v_j\rangle$ 上的权值，n 为网中顶点数。

显然，这是一个从源点开始的递推公式，Ve(j)的计算必须在顶点 v_j 的所有前驱顶点的最早发生时间全部求出后才能进行；所以，必须对 AOE-网进行拓扑排序，按拓扑有序序列逐个求出各个顶点所表示事件的最早发生时间。例如，图 8-29(a)中的顶点 v_4，其最早发生时间 Ve(4)是在求得 Ve(1)＝0，Ve(2)＝3，Ve(3)＝2 之后，才可求得：

$$Ve(4)=\max\{Ve(2)+3, Ve(3)+4\}=6$$

2. 事件的最晚发生时间 Vl(i)

Vl(i)是指在不推迟整个工程完成日期的前提下事件 v_i 所允许的最晚发生时间。对于一个工程来说，计划用几天完成是可以从 AOE-网中求得的，其数值就是汇点 v_n 的最早发生时间 Ve(n)，而这个时间也就是 Vl(n)。其他顶点事件的最晚发生时间应从汇点开始，逐步向源点方向递推才能求出，所以 Vl(i)的计算公式应是：

$$Vl(n)=Ve(n)$$

$$Vl(i)=\min\{Vl(j)-dut(\langle v_i,v_j\rangle)\} \quad \langle Vi,Vj\rangle\in S, 1<=i<=n-1$$

其中，S 是所有从 v_i 发出的弧的集合。

显然，Vl(i)的计算必须在 v_i 的所有后继顶点的最晚发生时间全部求出后才能进行；所以必须对 AOE-网进行逆拓扑排序，然后按逆拓扑有序序列进行递推求出各顶点事件的最晚发生时间。例如，图 8-29(a)中的顶点 v_2，其 Vl(2)是在先求出 Vl(6)＝8，Vl(5)＝7，Vl(4)＝6 之后才可求得：

$$Vl(2)=\min\{Vl(5)-3, Vl(4)-2\}=4$$

3. 活动的最早开始时间 Ee(i)

Ee(i)是指 a_i 所表示的活动最早可以开工的时间。若活动 a_i 是由弧 $\langle v_j,v_k\rangle$ 表示，则 Ee(i)＝Ve(j)。这说明，活动 a_i 的最早开工的时间等于事件 v_j 最早发生时间。

4. 活动的最晚开始时间 El(i)

El(i)是指在不推迟整个工程完成日期的前提下，活动 a_i 最晚允

许开始时间。若活动 a_i 由弧 $\langle v_i, v_k\rangle$ 表示，则：

$$El(i)=Vl(k)-dut(\langle v_i, v_k\rangle)$$

对于活动 a_i 来说，El(i)－Ee(i)表示完成活动 a_i 的时间余量。若 Ee(i)＝El(i)，表明该活动的最早开工日期同系统允许该活动的最迟开工日期相等；也就是说如果活动 a_i 不能按时完工，则整个工程就要延期，所以它是一个关键活动。显然，求出网中的所有关键活动之后，也就是求出了关键路径。

根据上面的分析，求 AOE-网的关键路径的算法可按以下步骤进行：

(1) 从源点 v_1 出发，令 Ve(1)＝0，按拓扑有序序列求其余各顶点的最早发生时间 Ve(i) (2<＝i<＝n)。如果得到的拓扑有序序列中顶点个数小于网中顶点数 n，则说明网中存在环，不能求关键路径，算法终止；否则执行步骤(2)。

(2) 从汇点 v_n 出发，令 Vl(n)＝Ve(n)，按逆拓扑有序序列求其余各顶点的最迟发生时间 Vl(i) (1<＝i<＝n－1)。

(3) 根据各顶点的 Vl 和 Ve 值，求出每个活动 a_i 的最早开始时间和最晚开始时间，同时若某活动 a_i 满足 Ee(i)＝El(i)。则 a_i 为关键活动。依次输出这些关键活动。

例如，图 8-29(a)所示的网，其按上述算法的计算结果如图 8-30 所示。可见 a_2，a_5 和 a_7 为关键活动，组成一条从源点到汇点的关键路径，如图 8-29(b)所示。

顶点	Ve	Vl	活动	Ee	El	El－Ee
V_1	0	0	a_1	0	1	1
V_2	3	4	a_2	0	0	0
V_3	2	2	a_3	3	4	1
V_4	6	6	a_4	3	4	1
V_5	6	7	a_5	2	2	0
V_6	8	8	a_6	2	5	3
			a_7	6	6	0
			a_8	6	7	1

图 8-30 AOE-网中顶点的发生时间和活动的开始时间

习 题

1. 设一个有向图的邻接矩阵如下：

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

(1) 画出该有向图；

(2) 求每个顶点的入/出度；

(3) 画出邻接表；

(4) 画出逆邻接表；

(5) 求强连通分量。

2. 编写在有 n 个顶点的有向图的邻接表上，统计每个顶点的入度、出度和度数的算法。

3. 试利用栈编写按深度优先搜索策略，遍历一个强连通图的非递归算法，要求图用邻接表作为存储结构。

4. 设一个无向带权图的耗费矩阵如下所示，请写出它的邻接表，并用克鲁斯卡尔算法求其最小生成树。

$$\begin{bmatrix} \infty & 2 & 3 & \infty & \infty & \infty & \infty & \infty \\ 2 & \infty & \infty & 2 & \infty & \infty & \infty & \infty \\ 3 & \infty & \infty & 1 & \infty & \infty & \infty & \infty \\ \infty & 2 & 1 & \infty & 2 & 4 & \infty & \infty \\ \infty & \infty & \infty & 2 & \infty & 1 & 2 & \infty \\ \infty & \infty & \infty & 4 & 1 & \infty & 2 & 1 \\ \infty & \infty & \infty & \infty & 2 & 2 & \infty & 3 \\ \infty & \infty & \infty & \infty & \infty & 1 & 3 & \infty \end{bmatrix}$$

5. 以邻接表作存储结构实现求从源点到其余各顶点的最短路径的 Dijkstra 算法。

第九日　查　　找

在本书的第三日到第八日中，我们讨论了各种线性或非线性的数据结构及其应用。在第九日和第十日中，我们将介绍应用数据结构来解决实际问题时，经常遇到的两种技术——查找和排序。

查找就是检索，亦即查表。查表是由同一类型的数据元素（或称记录）组成的集合；集合中的每个记录有若干个域组成，其中用于区分表中各个记录的域称为关键字域，其值称为关键字。所谓查找就是根据给定的某个值 K，在查找表中寻找关键字值等于 K 的记录的过程。若表中存在这样的记录，也就是说找到这样的记录，则称查找成功，此时查找的结果为给出整个记录的信息，或指示该记录在查找表中的位置；若表中不存在关键字值等于给定值 K 的记录，则称查找不成功，此时查找的结果可给出“空”记录或“空”指针。

查找的方法很多，对于不同的结构需采用不同的方法。本日主要介绍静态查找表、动态查找表和哈希表的查找。

9.1　静态查找表

在静态查找表中，记录按其逻辑顺序存放在计算机存储器中，并且在查找过程中，表中元素不会发生变化。静态查找表可以有不同的表示方法；在不同的表示方法中，其查找操作的实现方法也不同。

一、顺序表的查找

若静态查找表采用顺序存储结构，则称为顺序表；此时查找操作可用顺序查找来实现。顺序查找是最简单的一种查找方法，它是用给

定的值与表中各个记录的关键字逐个进行比较，若某个记录的关键字值和给定值相等，则查找成功；反之，若找遍表中所有记录，其关键字值和给定值均不等，则表明表中没有所查记录，查找不成功。下面，我们在顺序存储结构上讨论这种查找方法的实现。图 9-1 是顺序查找算法的框图描述。

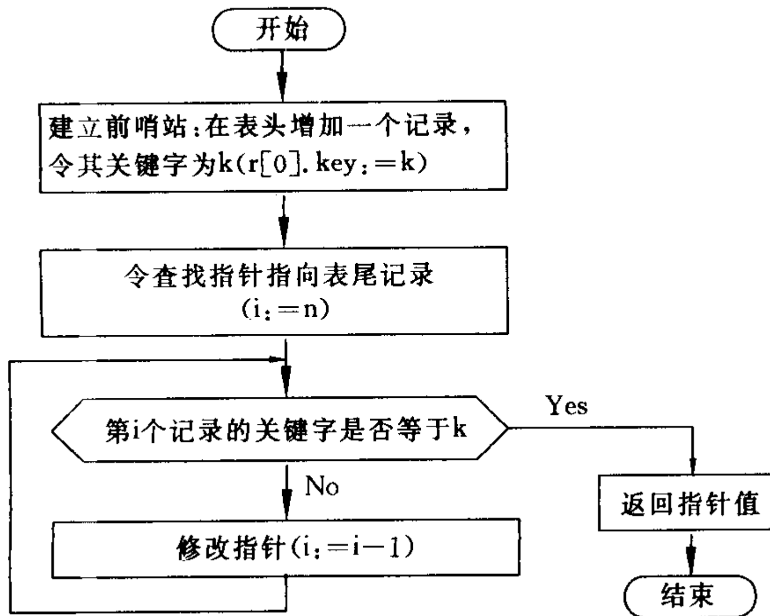


图 9-1 顺序查找算法的框图

算法的 PASCAL 语言描述如下：

```
CONST nmax={表中记录的最大数目}；
TYPE node=RECORD
              key:integer;
              ch:char
             END;
      sqlisttp= ARRAY [0..nmax] OF node;
FUNCTION seqsrch(r:sqlisttp; n,k:integer):integer;
  VAR i:integer;
  BEGIN
    r[0].key:=k; i=n;
    WHILE r[i].key<>k DO i:=i-1;
    return(i)
```

END;

运行这个算法，在查找成功时，将返回所找到记录在向量中的下标；否则，返回零。在查找之前先对 r[0]的关键字赋值 K，目的在于免去查找过程中每一步都要检测整个表是否已查找完毕。r[0]起到了监视哨的作用，这仅是一个程序设计技巧上的改进；然而实践证明，这个改进能使顺序查找在 n>=1000 时，进行一次查找所需的平均时间几乎减少一半。当然，监视哨也可设在高下标处。

在前面几日中，分析算法时，主要分析算法的时间复杂度和空间复杂度。对于查找算法，其执行时间主要取决于给定值同关键字的比较次数。所以，在本日以平均比较次数作为衡量查找算法好坏的依据之一。对于含有 n 个记录的顺序表，查找成功时平均查找长度为：

$$ASL = \sum_{i=1}^{n} P_i C_i$$

其中：P_i 为查找表中第 i 个记录的概率，C_i 为查找第 i 个记录时所需的比较次数。

对于顺序查找算法，假设每个记录的查找概率相等，即 $P_i = 1/n$ $(i=1,\cdots,n)$；而 C_i 取决于所查记录在表中的位置。如查找 r[n]时仅需一次比较，而查找 r[1]时需要 n 次比较。一般情况下，C_i 为 $n-i+1$，所以顺序查找成功的平均查找长度为：

$$\begin{aligned} ASL &= n * p_1 + (n-1) * p_2 + \cdots + P_n \\ &= 1/n(n+(n-1)+\cdots+1) \\ &= (n+1)/2 \end{aligned}$$

而查找不成功的比较次数总为 n+1，所以，顺序查找的时间复杂度为 O(n)。

显然，当静态查找表采用线性链表存储结构时，上述顺序查找方法的思想也是适用的。

二、有序表的查找

在静态查找表中，如果各记录的次序是按其关键字的大小顺序（比如，从小到大的顺序）排列的，则称为有序表。对顺序存储的有序

表(即有序的顺序表)可采用折半查找。

折半查找的基本思想是:先取表的中间位置记录的关键字同给定值进行比较,如果给定值与该记录的关键字值相等,则查找成功;否则,分二种情况,分别在表的前半部分或后半部分继续进行折半查找(当给定值小于中间记录的关键字值时,在前半部分进行查找;否则,在后半部分进行查找)。如此反复,直到找到或者查找区间小于零为止。

例如:有如下 11 个记录的有序表:

(8,15,18,23,41,56,63,79,82,90,93)

现要查找关键字为 18 和 80 的记录。

设指针 low 和 hig 分别指示待查记录所在区间的下界和上界,指针 mid 指向区间的中间位置,即 mid=(low+hig)/2。查找记录关键字 K=18 的过程为:

8 15 18 23 41 56 63 79 82 90 93

↑ ↑ ↑

low mid hig

此时 r[mid].key>k,说明如果待查记录存在,必在区间[low,mid-1]中,故令 hig=mid-1。

8 15 18 23 41 56 63 79 82 90 93

↑ ↑ ↑

low mid hig

这时,r[mid].key=k,说明查找成功。mid 就为该记录在向量中的位置。下面再看查找记录关键字 K=80 的过程。

8 15 18 23 41 56 63 79 82 90 93

↑ ↑ ↑

low mid hig

r[mid].key<k,故令 low=mid+1。

```
8   15   18   23   41   56   63   79   82   90   93
                              ↑         ↑         ↑
                             low       mid       hig
```

r[mid].key>k，令 hig=mid−1

```
8   15   18   23   41   56   63   79   82   90   93
                              ↑    ↑
                        low  mid  hig
```

r[mid].key<k，令 low=mid+1

```
8   15   18   23   41   56   63   79   82   90   93
                                   ↑
                             low  mid  hig
```

r[mid]<k。令 low=mid+1

```
8   15   18   23   41   56   63   79   82   90   93
                                   ↑    ↑
                                  hig  low
```

此时，因为 low>hig，说明表中没有关键字 K=80 的记录，查找不成功。

上述折半查找算法的框图描述如图 9-2 所示：

这个算法的 PASCAL 语言描述如下：

```
FUNCTION binsrch(r:sqlisttp; n,k:integer):integer;
VAR low,hig,mid:integer;
    found:boolean;
BEGIN
  low:=1; hig:=n; found:=false;
  WHILE NOT found AND low<=hig DO
    BEGIN
      mid:=(low+hig) DIV 2;
```

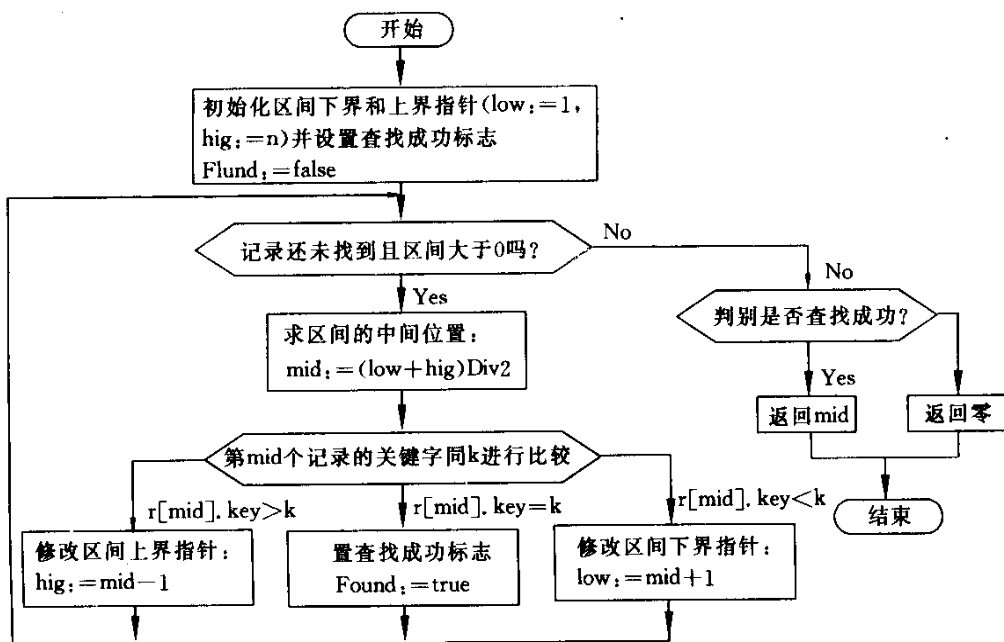


图 9-2　折半查找算法框图

```
            IF r[mid].key]>k
               THEN hig:=mid-1
               ELSE IF r[mid].key<k
                    THEN low:=mid+1
                    ELSE found:=true
        END;
      IF found THEN return(mid)
                ELSE return(0)
      END;
```

折半查找过程可用图 9-3 所示的二叉树来描述。二叉树中每个结点对应有序顺序表中的一个记录，结点中的值为该记录在表中的位置。从这棵二叉树可以看出，找到一个记录的过程恰好是走了一条从根结点到该记录对应结点的路径，和给定值进行比较的次数恰好为该结点在此二叉树上的层次数。例如，查找18 的过程恰好走了一条从根结点到结点③的路径，和给定值进行比

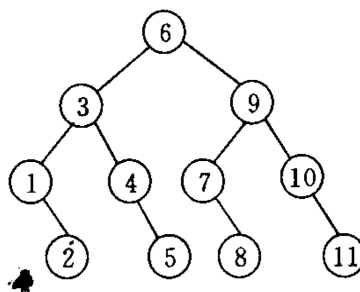


图 9-3　折半查找过程的二叉树描述

较的次数恰好为结点③在此二叉树上的层次数。所以，折半查找算法在查找成功时进行比较的次数最多不超过相应二叉树的深度，即不超过$\lfloor \log_2 n \rfloor+1$；这也是查找不成功时所用的最多比较次数。

在等概率的情况下，折半查找成功的平均查找长度为

$$ASL_{bs}=\sum_{i=1}^{n}P_iC_i=(n+1/n)\log(n+1)-1$$

显然，当 n 较大时，可取 $ASL_{bs}\approx\log2n$。可见，折半查找比顺序查找效率高；但折半查找只能适用于有序表，且限于顺序存储结构。

9.2 动态查找表

在这一节中，我们将讨论动态查找表的查找。动态查找表的特点是：表结构本身是在查找过程中动态生成的，即对于给定值 K，若表中存在其关键字等于 K 的记录，则查找成功返回；否则，插入关键字等于 K 的记录。动态查找表可有不同的表示方法；显然，不管采用什么表示方法，应要求能便于进行插入和删除操作。在本节中将讨论当动态查找表以各种树结构表示时的查找方法。

一、二叉排序树和平衡二叉树

1. 二叉排序树

二叉排序树或者是一棵空树，或者是具有下列性质的二叉树：(1)若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值；(2)若它的右子树不空，则右子树上所有结点的值均大于等于它的根结点的值；(3)它的左、右子树也分别为二叉排序树。例如，图 9-4 所示为两棵二叉排序树。

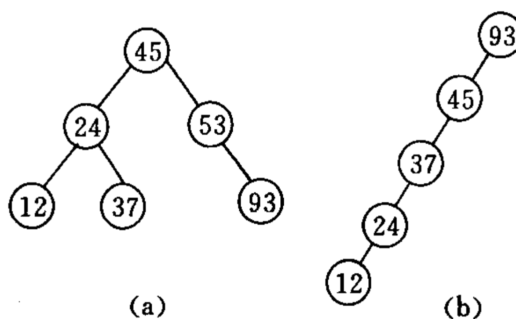


图 9-4　二叉排序树示例

二叉排序树又称二叉查找树，根据上述定义，可以按照如下过程

进行查找：首先将给定值与根结点的关键字进行比较，若相等，则查找成功；否则，将依据给定值同根结点的关键字之间的大小关系，分别在左子树或右子树上继续进行查找。显然，这是一个递归算法。图9-5为其框图描述。

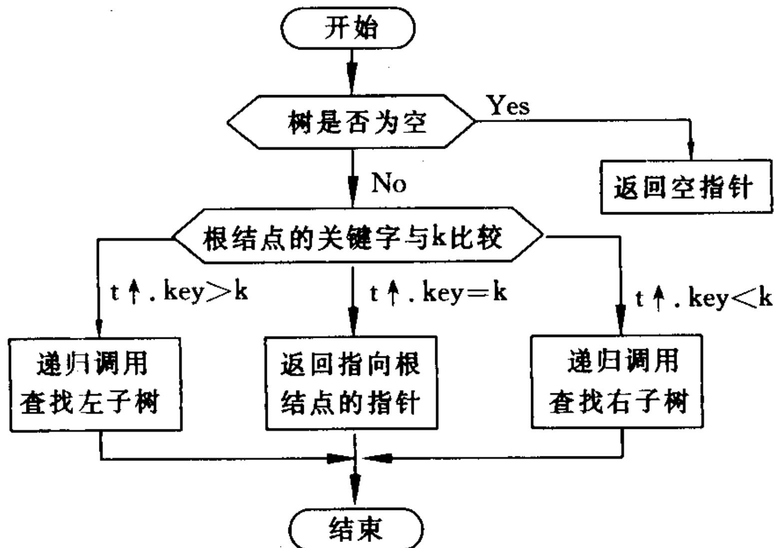


图 9-5　二叉排序树查找算法框图

其 PASCAL 语言描述如下：

```
TYPE bitreptr= ↑node;
      node=record
            key:integer;
            lchild,rchild:bitreptr
          END;
FUNCTION bstsrch(t:bitreptr; k:integer):bitreptr;
BEGIN
  IF (t=nil) OR (t= ↑.key=k)
    THEN return(t)
    ELSE IF t↑.key>k
          THEN return(bstsrch(t↑.lchild,k))
          ELSE return(bstsrch(t↑.rchild,k))
END;
```

例如，在图9-4(a)所示的二叉排序树中查找关键字值等于37的记录，其过程为：首先以根结点的关键字同37进行比较，因为

37<45，则查找 45 的左子树，此时左子树不空，且 37>24。则继续查找 24 的右子树，由于 37 和 24 的右孩子的关键字值相等，所以查找成功，返回指向结点 37 的指针。又如，在图 9-4(a)中查找关键字值为 50 的记录，同上述过程一样，在 50 先后与结点关键字值 45、53 比较之后，继续查找结点 53 的左子树，由于结点 53 的左子树为空，说明树中没有待查记录，所以查找不成功，返回指针为“nil”。

接下来我们要讨论的一个问题是：如何建立二叉排序树？

对 n 个记录的关键字序列 K，$K=\{k_1,k_2,\cdots,k_n\}$，当要建立二叉排序树时，就从一个空的二叉排序树开始，将记录逐个插入到这棵二叉排序树中，插入的次序就按 k_i 的下标 i 的顺序进行，具体步骤如下：

(1) 令关键字为 k_1 的记录结点为二叉排序树的根；

(2) 若 $k_2<k_1$，则令 k_2 的记录结点作为 k_1 的左子树的根结点；否则，令 k_2 的记录结点作为 k_1 的右子树的根结点。

(3) 对 $k_3,k_4,\cdots,k_n$ 重复步骤(2)的操作。

例如，关键字序列 k={12,20,5,10,21,4,9,10}，其建立二叉排序树的过程如图 9-6(a)～(h)所示。

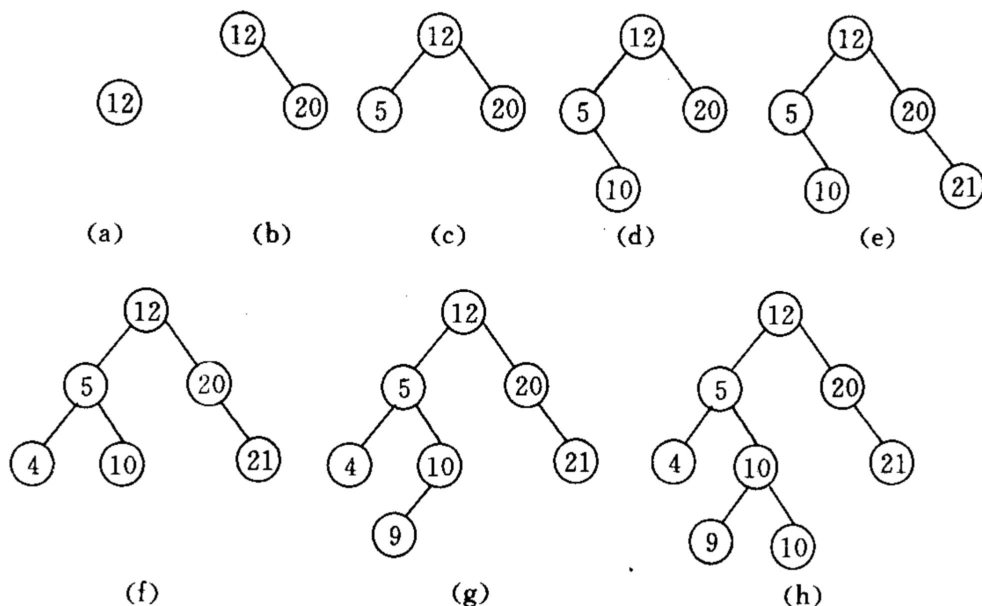


图 9-6 建立二叉排序树的示例

根据以上建立二叉排序树的规则可以很容易地写出一个递归的算法。下面我们将介绍在二叉排序树上插入结点的非递归算法，其算法框图如图 9-7 所示：

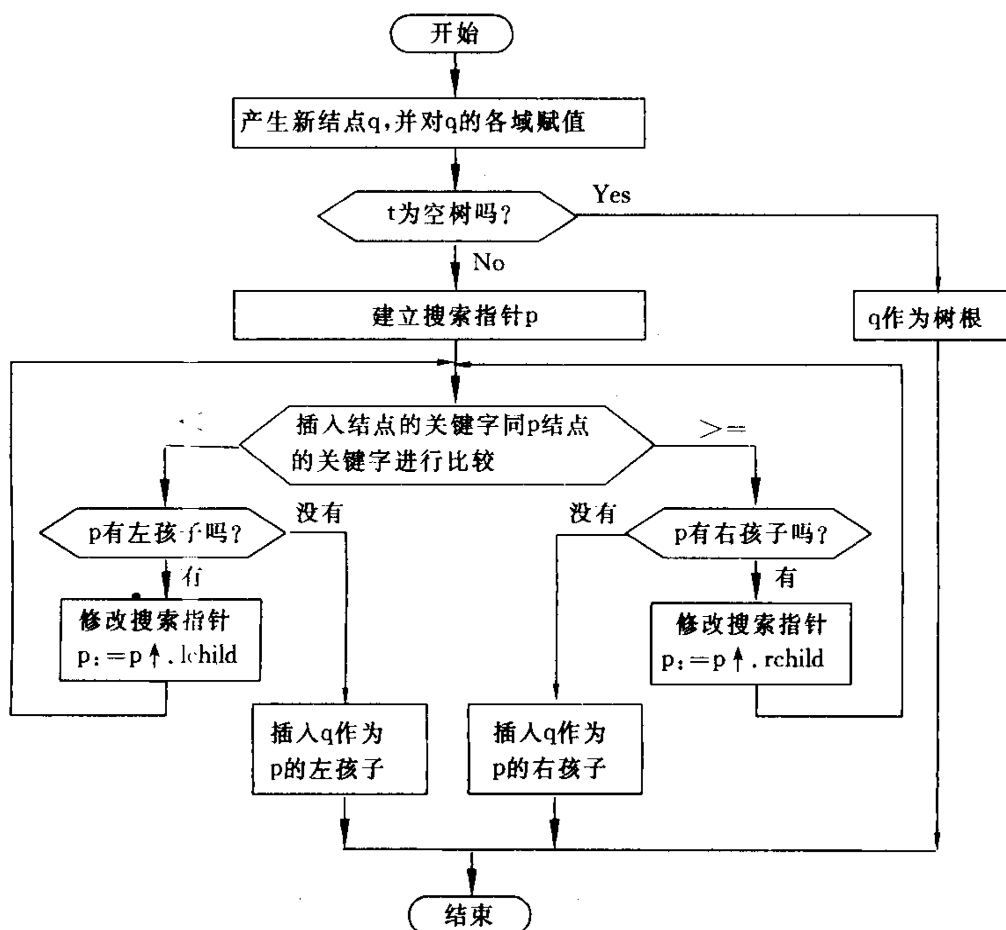


图 9-7 二叉排序树中插入结点的算法框图

其 PASCAL 语言描述如下：

```
PROCEDURE ins-bstree(VAR t:bitreptr; x:integer);
VAR p,q:bitreptr;
     bool:boolean;
BEGIN
  new(q); q↑.key:=x; q↑.lchild:=nil; q↑.rchild:=nil;
  IF t=nil THEN
    t:=q
   ELSE BEGIN
        p:=t; bool:=false;
```

```
        REPEAT
          IF x<p↑.key THEN
            IF p↑.lchild<>nil
              THEN p:=p↑.lchild
              ELSE BEGIN
                  p↑.lchild:=q;
                  bool:=true
                END
            ELSE IF p↑.rchild<>nil
                THEN p:=p↑.rchild
                ELSE BEGIN
                    p↑.rchild:=q;
                    bool:=true
                  END
          UNTIL bool=true
        END
  END;
```

过程 ins_bstree 实现了在二叉排序树上插入一个结点，反复调用这个过程就可以建立二叉排序树。容易看出，中序遍历二叉排序树可得到一个记录按关键字的有序序列。这就是说，一个无序序列可以通过构造一棵二叉排序树而变成一个有序序列；并且，从上面的插入过程还可以看到，每次插入的新结点都是作为二叉排序树上新的叶子结点，所以在进行插入操作时，不必移动其它结点。二叉排序树既拥有类似于折半查找的特性，又能方便地进行插入，因此不失为动态查找表的一种适宜表示。

在二叉排序树上删除一个结点也很方便。当然，删除某个结点之后必须依旧保持二叉排序树的特性。假设在二叉排序树上被删除结点由 P 所指示，其双亲结点由 f 指示，且不失一般性，可设 p 为 f 的左孩子，则删除结点 p 的操作可分如下四种情况进行。

(1) p 结点为叶子，即 p 没有左、右孩子，则可直接删去 p。如图 9-8(a)所示。

(2) p 结点无左子树，而有右子树，则可用 p 的右孩子取代 p。如图 9-8(b)所示。

(3) P 结点有左子树，而无右子树，则可用 p 的左孩子取代 p。如图 9-8(c)所示。

(4) p 结点的左、右子树都存在，则可以先找到左子树中关键字最大的结点 s，然后用 s 来取代结点 p，并且在左子树中删除 s 结点。如图 9-8(d)所示。

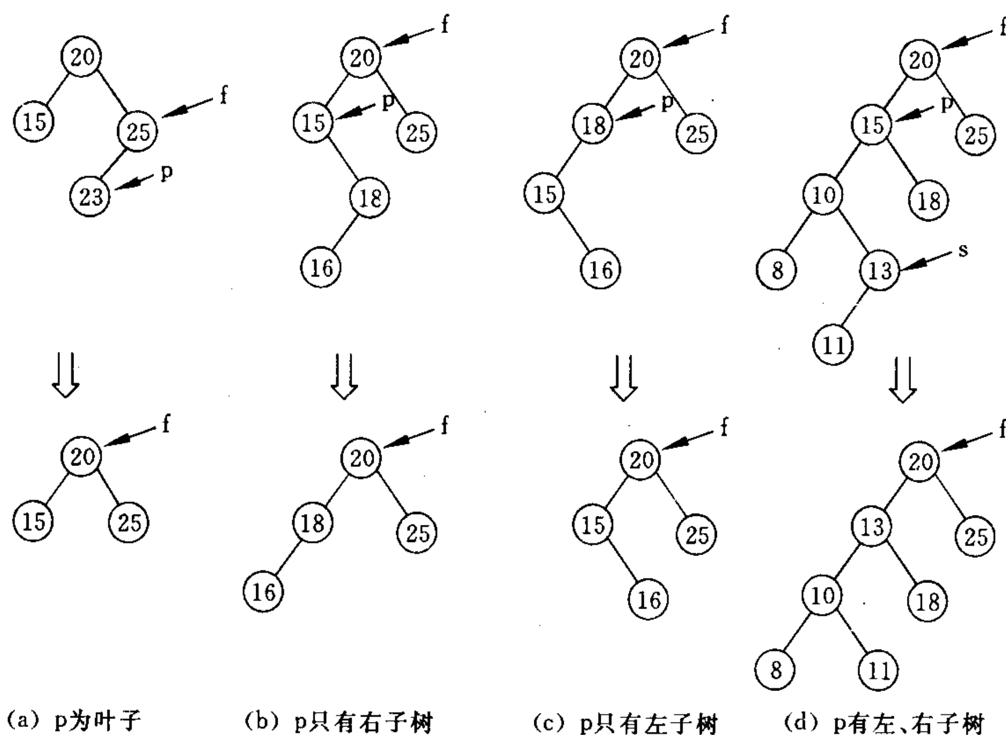


图 9-8　二叉排序树中删除结点示例

综合以上四种情况，读者可以自已写出在二叉排序树上删除一个结点的完整算法。

分析二叉排序树查找算法可以看出，在二叉排序树上查找其关键字值等于给定值的过程，恰是走了一条从根结点到该结点的路径，所以，同折半查找类似，查找时同给定值的比较次数不超过树的深度。但含有 n 个记录的二叉排序树是不唯一的。例如，图 9-4(a)、(b)所示的二棵二叉排序树中结点的值都相同，但前者由关键字序列(45,24,53,12,37,93)构成，而后者由关键字序列(93,53,45,37,24,12)构成。(a)的深度为 3，而(b)的深度为 6。再从平均查找长度来看，

假设 6 个记录的查找概率相等，均为 1/6，则(a)的平均查找长度为：

$$ASL(a)=(1+2+2+3+3+3)/6=14/6$$

而(b)的平均查找长度为：

$$ASL(b)=(1+2+3+4+5+6)/6=21/6$$

所以，含有 n 个结点的二叉排序树的平均查找长度和二叉排序树的形态有关。当先后插入的记录按关键字有序时，构成的二叉排序树将蜕变成为单支树，其深度为 n，平均查找长度为(n+1)/2，此时它和顺序查找一样，这是最差的情况。最好情况是，当生成的二叉排序树中任一结点的左、右子树的深度相差不超过 1 时，其平均查找长度同折半查找相同。所以，为了在所构成的二叉排序树上能有效地进行查找，就要在构造过程中进行"平衡化"处理，使之成为平衡二叉树。

2. 平衡二叉树

平衡二叉树又称 AVL 树。它或者是一棵空树，或者是具有下列性质的二叉树：(1)它的左子树和右子树都是平衡二叉树；(2)其左、右子树深度之差的绝对值不超过 1。在这里，我们对二叉树上的每个结点定义一个平衡因子(bf)，其值为该结点的左子树的深度减去它的右子树深度。显然，平衡二叉树上所有结点的平衡因子只可能是 −1、0 和 1。只要二叉树上有一个结点的平衡因子的绝对值大于 1，则该二叉树就不是平衡的。如图 9-9(a)为一棵平衡二叉树，而图 9-9(b)为一棵不平衡的二叉树(结点中的值为该结点的平衡因子)。

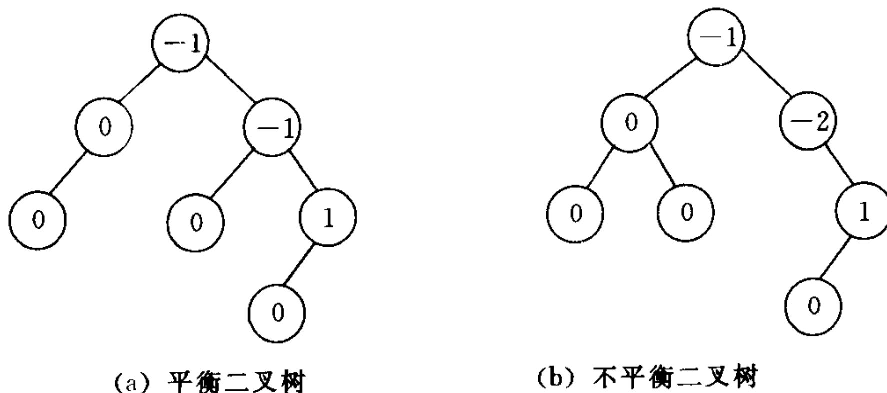


图 9-9　平衡二叉树和不平衡二叉树示例

我们希望由任何关键字序列构成的二叉排序树都是 AVL 树。

因为 AVL 树上任何结点的左右子树的深度之差都不超过 1，所以其平均查找长度和 logn 同数量级。

那么，如何使构造的二叉排序树为平衡二叉树呢？下面我们先看一个具体例子。假设表中关键字序列为(15,27,33,94,68)。显然，空树和由一个结点 15 组成的树都是平衡二叉树。在插入 27 之后仍是平衡的，只是根结点的平衡因子 bf 由 0 变为－1。在继续插入 33 之后，由于结点 15 的 bf 值由－1 变成－2，出现了不平衡现象。此时，可以对树作一个逆时针"旋转"，令结点 27 为根，而结点 15 为它的左子树，如图 9-10(d)～(e)所示。继续插入 94 和 68 之后，由于结点 33 的 bf 值变成-2，二叉排序树中又出现了新的不平衡现象，需进行调整。但此时由于结点 68 插在结点 94 的左子树上，因此不能如上作简单调整。对于以结点 33 为根的子树来说，既要保持二叉排序树的特性，又要平衡，则必须以结点 68 作为根结点，而使 33 成为它的左子树的根，94 成为它的右子树的根。这好比对树作二次"旋转"操作——先顺时针旋转，然后逆时针旋转，如图 9-10(f)～(h)所示：

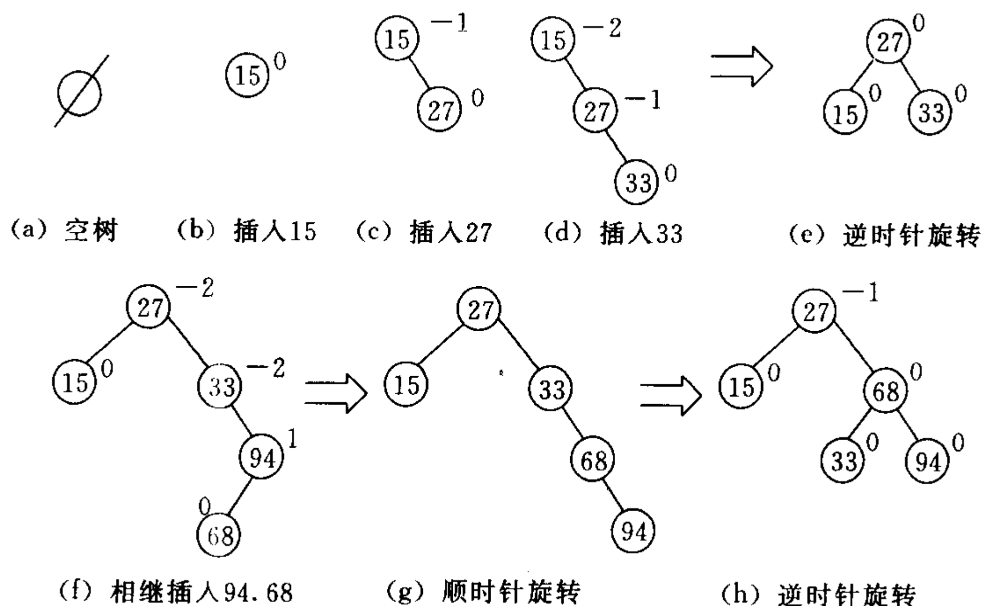


图 9-10 平衡二叉树的生成过程

一般情况下，假设由于在平衡二叉树上插入结点而失去平衡的最小子树的根结点指针为 A(即 A 是离插入结点最近，且平衡因子

绝对值超过1的结点),则失去平衡后进行调整的规律可归纳为下列四种情况:

(1) LL型平衡旋转:由于在A的左孩子的左子树上插入结点,而使A的平衡因子由1变成2导致二叉树失去平衡。这时,需进行一次顺时针旋转操作,如图9-11(a)所示。

(2) RR型平衡旋转:由于在A的右孩子的右子树上插入结点,而使A的平衡因子由-1变成-2导致二叉树失去平衡。这时,需进行一次逆时针旋转操作,如图9-11(b)所示。

(3) LR型平衡旋转:由于在A的左孩子的右子树上插入结点,而使A的平衡因子由1变成2导致二叉树失去平衡。这时,需进行二次旋转(先逆时针,后顺时针),如图9-11(c)所示。

(4) RL型平衡旋转:由于在A的右孩子的左子树上插入结点,而使A的平衡因子由-1变成-2导致二叉树失去平衡。这时,需进行二次旋转(先顺时针,后逆时针),如图9-11(d)所示。

具体的平衡算法,这里就不作讨论了,读者可以根据上述四种情况进行考虑。

二、B树

前面介绍的查找方法,均适用于存储在计算机内存中的文件或表,统称为内查找方法。若要查找在外存储器(如磁盘)上的信息时,则要用外查找方法。在此介绍的B树就是其中之一。

1970年BaYer提出了一种多叉平衡树,称为B树。一个m阶的B树满足下述条件:

(1) 每个结点至多有m个孩子;

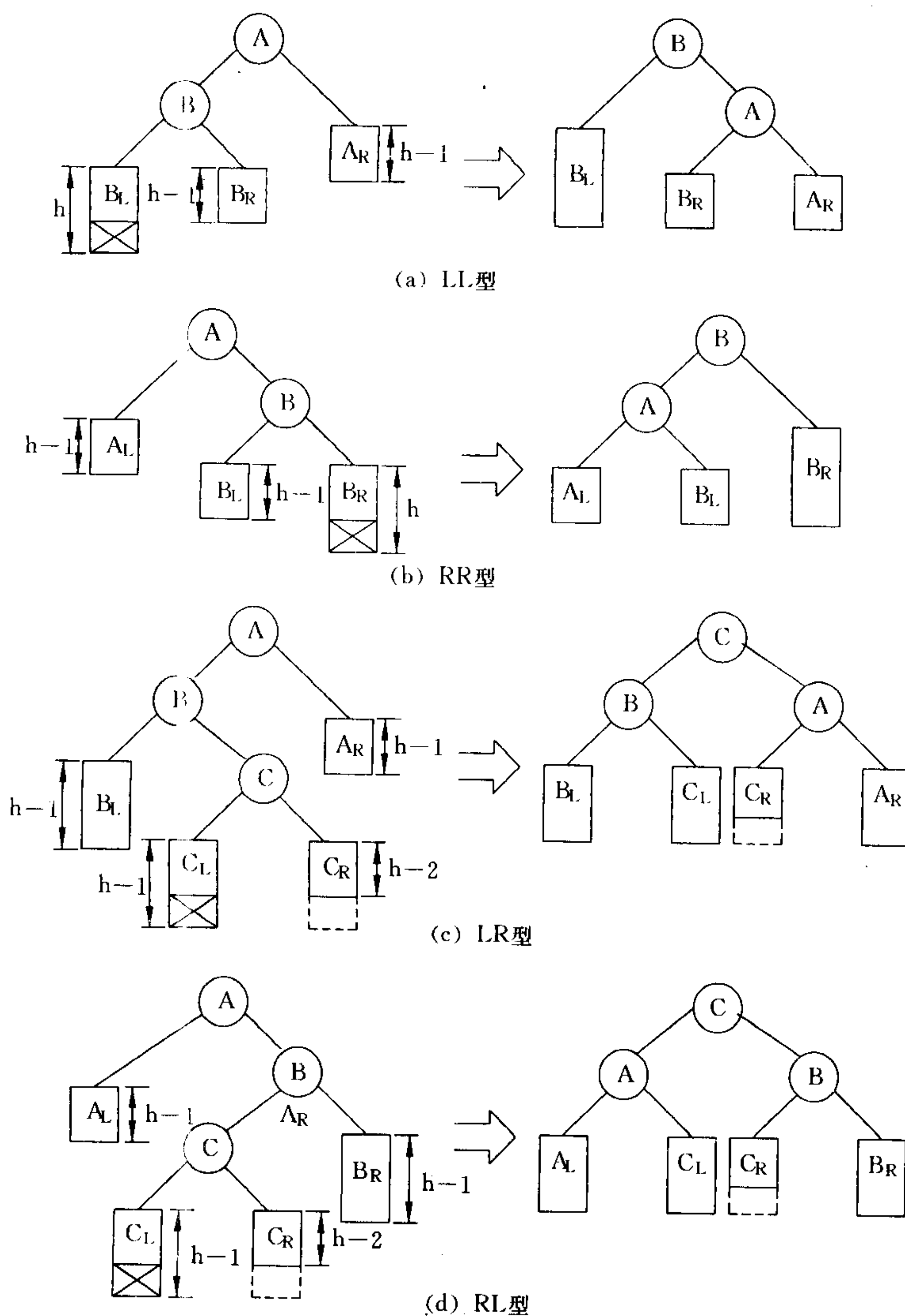
(2) 除根结点和终端结点(树叶)之外,每个结点至少有$\lfloor m/2 \rfloor$个孩子;

(3) 根结点至少有两个孩子(除非它本身又是树叶);

(4) 具有n个孩子的非终端结点含有n-1个关键字;

(5) 所有的终端结点都出现在同一层上,并且都不带信息。

图9-12给出了一个含有j个关键字和j+1个指针的B树结点。



(a) LL型

(b) RR型

(c) LR型

(d) RL型

图 9-11　二叉平衡树的平衡旋转图示

该结点中的关键字满足下列条件：

$K_1 < K_2 < \cdots < K_j$，指针 P_i 指向一个子树的根，该子树中所有结点所含的关键字均大于 K_i，小于 K_{i+1}。

在B树上查找的过程如下：设如图9-11所示的结点p已从外存读到内存中。我们可以选择适当的内查找法在结点p中查：若j比较大时，可选用折半查找；若j较小，则可采用顺序查找。现给定一个关键字x，若x在结点p中，则查找成功；否则，必是下述三种情况之一：

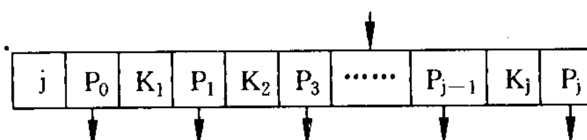


图9-12 B树结点

(1) $K_i < x < K_{i+1}$，其中$1 <= i < j$，则沿指针P_i继续查。

(2) $x > K_j$，则沿指针P_j继续查。

(3) $x < K_1$，则沿指针P_0继续查。

如果在某种情况下，相应指针指向终端结点(即叶子)，则查找失败，说明关键字x不在该B树中。

当查找不成功时，需要进行插入。显然，每次插入总是从最下层的非终端结点开始的。对于一棵m阶的B树，若在一个不满m－1个关键字的结点中插入，则可以直接进行。若在一个已有m－1个关键字的结点中进行插入，则要将该结点“分裂”成两个结点，从而还需对它的上层结点进行插入。现举例说明。

图9-13(a)所示是一棵5阶B树。当要在该树中插入关键字为36，37的记录时，可以直接插在结点p中。然后，当继续要求插入关键字35时，由于p结点已满，则要将它“分裂”成两个结点，按平均分配的方式将关键字分配给这两个结点。这时需将35插入到上层结点中，因上层结点已填满，故再次形成“分裂”，将40插入到更上一层的结点中，最后的情况如图9-13(b)所示。

从上例可以看出，在B树中插入时，产生结点“分裂”是由下层结点逐步向上层传递的；在极端情况下，会一直传递至根结点。实际上，这也是B树增高的唯一途径。

若要在B树中删除关键字，其处理过程比插入复杂。先要找到被删关键字K_i的位置，若K_i在最下层的非终端结点中，则可直接删去；否则，当删去K_i后，其空出的位置要抽调P_i所指子树中的最小

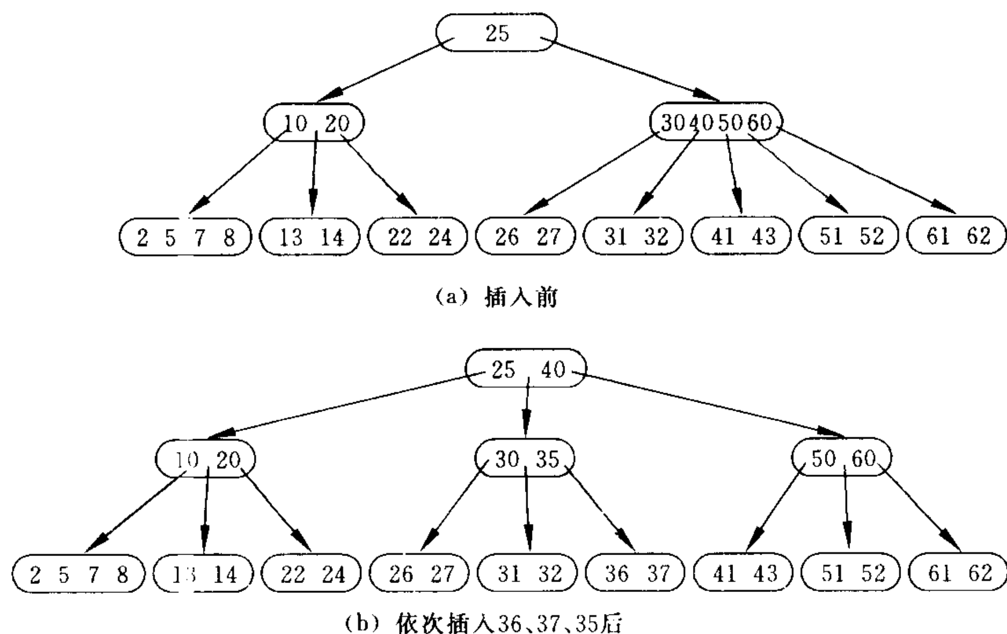


图 9-13　在 B 树中插入结点

关键字来填补(这个最小关键字一定在最下层的一个非终端结点中)。另外,还需检查当结点中关键字因删除、抽调而减少后,是否还符合 B 树的定义;若不符合,则要作结点"合并"等相应的处理。

9.3　哈希表查找

一、哈希表

哈希表的查找思想与前面介绍的查找方法完全不同。无论是顺序查找、折半查找还是二叉排序树查找,都要通过一系列的关键字比较才能确定被查记录在表中的位置;所以,这类方法统称为对关键字进行比较的查找方法。而哈希法却是利用关键字进行转换,计算出记录存放地址的查找方法。

如果一个关键字对应一个地址,这就是最直观,最简单的哈希法;但这种方法往往行不通。例如,有一个符号表其标识符至多由五个字母组成,则可能有

$$26^5+26^4+26^3+26^2+26=12356630$$

个不同的标识符，也就是说可能有这么多个不同的关键字；如果一个关键字对应一个地址，那其存储空间就难以满足要求。因此，这样的一一对应关系虽然简单、方便，但实用价值不大；而且在实际问题中，一般不可能有这么多的标识符同时出现在一个源程序中。那么，是否可以设想一种方法，既能利用关键字直接找到其存储位置，又不致于占用太多的存储空间而达到存储符号表的目的。显然，这一设想将导致一个问题，即由于希望尽量减少存储空间，所以会造成两个不同的关键字被转换到同一个存储地址上去，此时叫做发生冲突。另外，如何利用关键字直接转换成存储地址呢?这需要设计一个函数，其自变量是关键字，这个函数称为 Hash 函数，用 H 表示。该函数把变化范围很广并可识别的关键字通过各种剪裁手段和简单的数学运算，转换成存储地址（或向量的标号），因此得名为“杂凑”。由于可以把 Hash 函数看成按照某种特定的方法将记录按关键字散列到内存储器的指定空间内，所以这种方法也称为散列地址法。另外，也可以称为关键字转换法。采用哈希法在连续的内存空间中建立起来的符号表，就称为哈希表。

哈希法可以这样来描述：对于关键字 k，可以定义一个简单的 Hash 函数 H(k)，使得 H(k)（或用 H(k)进行一些线性运算后）等于符号表存区内某一单元的地址；并且要求当 $K_1 \neq K_2$ 时，$H(K_1)=H(K_2)$的可能性尽量地小。

如果 $K_1 \neq K_2$ 而 $H(K_1)=H(K_2)$就是发生了冲突。实际上冲突是不可避免的，所以只要求发生冲突的可能性尽量地少。

因此，对于哈希表，主要研究下列两个问题：

1. 如何设计 Hash 函数？
2. 如何解决冲突？

二、构造哈希函数的基本方法

假定符号表的存储区是一个长度为 M 的向量，现采用 Hash 法进行存储，所要接收的不同关键字有 N 个，但不知道各种关键字出现的频率是多少。这时应如何构造 Hash 函数呢？

构造Hash函数的方法很多，衡量Hash函数好坏的主要标准是尽可能地减少冲突。要减少冲突，就要设法使哈希函数尽可能均匀地把关键字映射到符号表存区的各个存储地址上，这样可以提高查找效率。在构造哈希函数时，一般都要对关键字进行计算。为了尽量避免产生相同的哈希函数值，所以应使关键字的所有组成成份都能起作用。但关键字的组成是难以在事前全部确定的，因此，很难说那一种哈希函数是最好的。

符号表中各关键字是由字母组成的，在不同的计算机中都有相应的内部表示法，但最后总是以二进制或十进制的正整数来表示。例如，用二位数字的整数01～26表示对应的26个英文字母，则下列四个关键字与其内部的代码关系为：

关键字	KEYA	KEYB	AKEY	BKEY
内部代码	11052501	11052502	01110525	02110525

现以这四个关键字为例介绍几种常用的哈希函数。

1. 平方取中法

这是较常用的哈希函数，构造原则是：先计算出关键字内部代码的平方值，再取它的中间几位作为内存地址。对上面给出的四个关键字，其结果如下：

关键字	内部代码	内部代码平方值	哈希函数值
KEYA	11052501	122157773355001	773
KEYB	11052502	122157800460004	800
AKEY	01110525	001233265775625	265
BKEY	02110525	004454315775625	315

其哈希函数是取平方值的中间三位。如果符号表的存储地址是0～999，则上述哈希函数值就是存储地址。如果计算出的哈希函数值超过或不到存储区的地址范围，则需要乘一个比例因子，把哈希函数放大或缩小，使其落到符号表的存储区地址范围内。

2. 除留余数法

这种方法是用模(MOD)运算得到的。设给出的关键字为k，存

储区单元数为 m，则用一个小于 m 的质数 p 去除 k，得到余数 r，即：

$$r=k \text{ MOD } p$$

如果 r 落在存储区地址范围内，则 r 就取为哈希函数值；否则，再用一个线性函数求出哈希函数值。例如：有一组关键字从 000001 到 859999，指定的存储区地址为 1000000 到 1005999，即 m＝6000，可选 p＝5999，若要转换关键字 K＝172148，则有：

$$r=172148 \text{ MOD } 5999=4716$$

因 r 不在指定的地址范围内，所以取哈希函数为：

$$H=1000000+r$$

故有：

$$H(k)=H(172148)=1000000+172148 \text{ MOD } 5999=1004176$$

这样就把关键字 k 直接转换成存储地址了。

在此，p 的选择是很值得研究的。如果选择关键字内部代码的基数的幂次去除关键字，其结果必是关键字的低位数字，均匀性较差。如果取 p 为任意的偶数，则关键字内部代码为奇数时，其哈希函数值为奇数。因此，选 p 为偶数也不好。理论分析和试验结果均证明，p 应取小于存储区容量的素数。例如对前述的四个关键字，若符号表存区为 000 到 999，应取 p 为小于 1000 的素数，可取 p＝997，则可得以下结果：

关键字	内部代码	H(k)＝k MOD 997
KEYA	11052501	756
KEYB	11052502	757
AKEY	01110525	864
BKEY	02110525	873

这些结果是比较好的，所以除留余数法是经常使用的。

3. 数字分析法

对各个关键字内部代码的各个码位进行分析：第 i 位上，各个关键字中出现的数码种类比较多，则可选中该值为哈希函数值的一部分；需选多少位来组成哈希函数值应视存储区地址范围而定。例如，需要哈希函数值（地址码）三位，则对下列关键字进行数字分析，其数

码种类较多的是第 4、8、9 三位，取这三位构成哈希函数值如下所示：

关键字	哈希函数值
000319426	326
000718309	709
000629443	643
000758615	715
000919697	997
000310329	329

这种方法比较简单、直观，但需要预先知道每个关键字的情况，这就限制了它的应用范围。

构造哈希函数的方法很多，除以上几种外，还有截段法，即截取关键字中的某一段数码作为哈希函数；分段迭加法，即把关键字的机内代码分成几段再进行迭加(可以是算术加，也可以是按位加)得到哈希函数值。对于各种构造哈希函数的方法，很难一概而论地评价优劣；任何一种哈希函数都应当用实际数据去测试它的均匀性，才能做出正确判断和结论。

三、解决冲突的几种方法

哈希法中不可避免地会出现冲突现象，所以应用哈希法时关键的问题是如何解决冲突。解决冲突的方法基本上有两大类：一类称为开放地址法，当发生冲突时，用某种方法形成一个探测的序列，沿着这个序列一个个单元地查询，直到找到这个关键字的记录或找到一个开放的地址(即没有进行存储的空单元)。此时，如果是插入操作，则遇到空单元就可以进行插入；如果是查找操作，则遇到空单元就是查找失败。另一类称为链地址法，当发生冲突时，就拉出一条链，建立一个链接方式的子表，使具有相同哈希函数值的关键字及其记录链接在同一个子表中。下面分别介绍这两类方法的具体算法。

1. 开放地址法

用开放地址法解决冲突时，要产生一个探测序列。最简单的产生



图 9-14 用开放地址法处理冲突示例

探测序列的方法是进行线性探测，即当发生冲突时，顺序地到存储区的下一个单元进行探测。假设某记录的关键字为 k，哈希函数 H(k)=j，若在 j 位置上发生冲突，则顺序地对 j+1 位置进行探测，依次类推。例如，在长度为 11 的哈希表中已填有关键字分别为 17、60、29 的记录(哈希函数为 H(key)=key MOD 11)，如图 9-14(a)所示。现有第四个记录需插入，其关键字为 38，由哈希函数得到其哈希地址为 5，产生冲突；若用线性探测再散列的方法处理，得到下一个地址 6；仍冲突；再求下一个地址 7，仍冲突；直到哈希地址为 8 的位置为“空”时，处理冲突的过程结束，记录填入哈希表中序号为 8 的位置，如图 9-14(b)所示。

2. 链地址法

链地址法是经常使用的处理冲突的方法，此方法很有效，它将所有哈希地址相同的记录存储在一个线性链表中。假设某哈希函数产生的哈希地址在区间[0..m－1]上，则设立一个指针型向量。

chainhash:ARRAY [0..m－1] OF pointer;

使其每个分量 chainhash[i]指向哈希地址为 i 的记录链表表头。当有哈希地址为 i 的记录插入时，可以插在由 chainhash[i]指向的链表表头，或表尾；也可以插在中间，以保持哈希地址相同的记录在线性链表中按关键字值有序。例如，已知一组关键字为(19,14,23,1,68,20,84,27,55,11,10,79)，则按哈希函数 H(key)=key MOD 13 和链地址法处理冲突所得到的哈希表如图 9-15 所示，同一链表中关键字自小至大有序。

3. 建立一个公共溢出区

在这种方法中，用两个表存储记录。假设哈希函数的值域为[0..m－1]，则设向量 hashtable[0..m－1]为基本表，每个分量 hashtable[i]存放一个哈希地址为 i 的记录；另设立一个向量 over[0..n]为溢

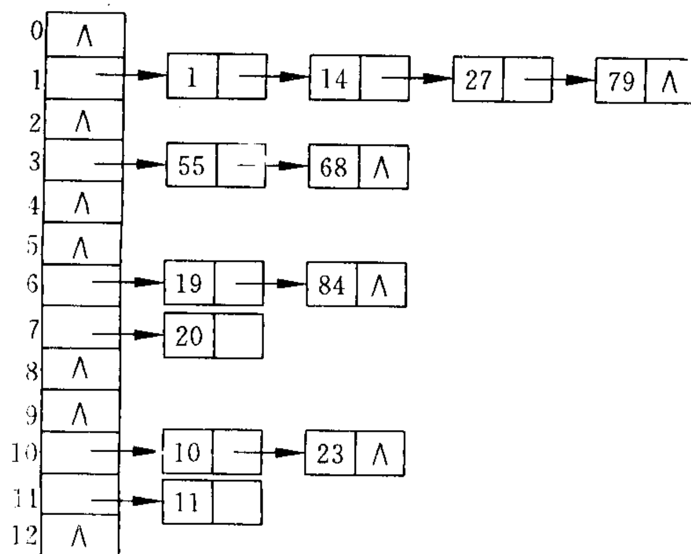


图 9-15 链地址法处理冲突时的哈希表

出表，它用于存储哈希地址发生冲突的记录。

四、哈希表的查找

在哈希表上进行查找的过程同哈希表的构造过程基本一致。设给定值为k，根据造表时设定的哈希函数求出哈希地址，若表中此位置上没有记录，则查找不成功；否则，比较关键字，若和给定值相等，则查找成功；否则说明发生冲突，根据造表时设定的处理冲突的方法找“下一地址”，直至哈希表中某个位置为“空”(查找不成功)，或者此位置记录的关键字等于给定值时为止(查找成功)。

设以开放地址法处理冲突，哈希函数以 hash 表示，则哈希表的查找过程的算法框图如图 9-16 所示：

假设在哈希表中，当某一位置上的 key 域值为零时，表示该位置未被占用，则此算法的 PASCAL 语言描述如下：

```
CONST
  mmax={哈希表长度}；
TYPE
  node=RECORD
         key:integer;
         ch:char
```

```
        END;
    hlistty=ARRAY [0..mmax-1] OF node;
  FUNCTION hashsrch(shtable:hlistty; k:integer):integer;
    VAR i,j:integer;
    BEGIN
      j:=hash(k); i=j-1;
      WHILE (shtable[j].key<>0) AND (j<>i) AND (shtable[j].key
<>k)
        DO j:=(j+1) MOD mmax;
      IF shtable[j].key=k
        THEN return(j)
        ELSE return(-1)
  END;
```

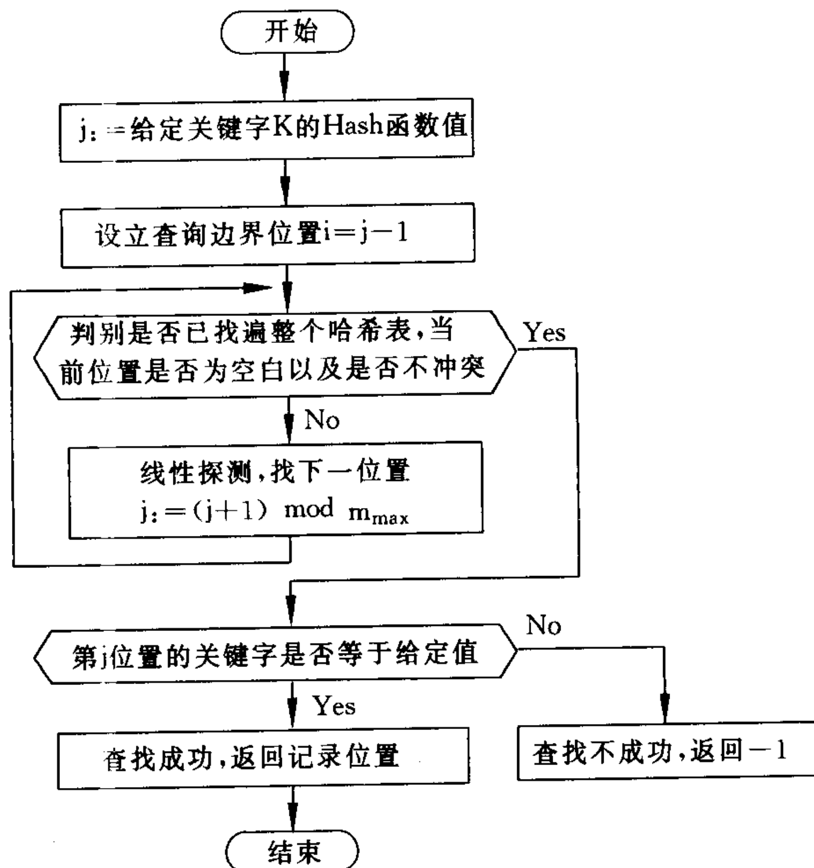


图 9-16　哈希表查找算法框图

下面对哈希法进行简单的分析。

哈希法是利用关键字进行转换计算后直接求出存储地址的。所以当由哈希函数能得到均匀的地址分布时，不需要进行比较就可以直接找到所查的记录。但实际上，冲突不可能完全避免冲突，因此查找时还需要进行探测、比较，查找的效率显然取决于发生(解决)冲突的次数。如果我们引进装填因子 α：

α=哈希表中的记录数/哈希表的长度

则 α 标志了哈希表装满的程度。直观地看，α 越小，发生冲突的可能性就越小；α 越大，即表中记录已很多，发生冲突的可能性就越大。

D. E. kunth 在"程序设计技巧"第三卷中指出：为了查找一个记录或插入一个新的记录，所需要的探测、比较次数仅依赖于装填因子 α。对于线性探测法，查找成功的平均查找长度为：$(1+1/(1-\alpha))/2$；查找不成功的平均查找长度为 $(1+1/(1-\alpha)^2)/2$。

值得提醒大家的是，若要在非链地址处理冲突的哈希表中删除一个记录，则需在该记录的位置上填入一个特殊符号，以免找不到在它之后插入的、在这个位置上发生过冲突的记录。

习　题

1. 试将折半查找的算法改写成递归调用形式的算法。
2. 试从空树开始，画出按以下次序向平衡二叉树插入关键字的建树过程：8，9，10，2，1，5，3，6，4，7，11，12。
3. 试写一个判别给定二叉树是否为二叉排序树的算法。二叉树以二叉链表作存储结构，且树中结点的关键字均不同。
4. 假设哈希表的长为 m，哈希函数为 H(x)，处理冲突的用链地址法。试编写输入一组关键字并构造哈希表的算法。

第十日　排　　序

在使用计算机进行数据处理时，经常要把数据按一定的规律排列起来。设有一组记录 $R_1, R_2, \cdots, R_n$，其对应的关键字为 $K_1, K_2, \cdots, K_n$。将此 n 个记录按其关键字大小递增(或递减)的次序排列起来，使得当 $i<j$ 时，$K_i <= (或>=) K_j$，这就称为排序(sorting)。排序在计算机软件中应用十分广泛；有时候排序是为了提高查找的速度，例如，折半查找的前提就是记录按关键字必须有序，有时候排序是系统管理上的需要。因此，为了提高计算机的工作效率，需要研究有效的排序方法。

排序的方法很多，根据记录所处的位置，可以分为内部排序和外部排序两大类。内部排序是指排序期间，全部数据都放在内存中进行的排序；外部排序是指当需要排序的记录非常之多，排序时全部记录已不能同时存入内存中，所以排序期间，不仅要使用内存，而且还需使用外部存储器的排序。本日主要讨论内部排序的一些常用方法。某些内部排序方法的思想是可以推广到外部排序的。

在本日讨论中，排序均以关键字从小到大的次序来进行。如果待排序的记录中，存在关键字相等的记录，当经过排序后，这些关键字相等的记录之间，相对次序保持不变，则称这种排序方法是稳定的，否则称为不稳定的。例如，有一组无序的关键字为：K_1、K_2、K_3、K_4、K_5，其中 $K_3=K_5$。若排序后，得到的有序序列为：K_3、K_5、K_4、K_2、K_1，因为 K_3 还是在 K_5 前面，所以这种排序方法是稳定的；若得到有序序列为：K_5, K_3, K_4, K_2, K_1，则改变了 K_3 和 K_5 的相对次序，所以这种排序方法是不稳定的。

在下面介绍的排序方法中，所用例子的数据都是指记录的关键

字；对关键字所施行的操作均是针对其相应的记录而言的。

10.1 插入排序

在本节中将介绍直接插入排序及其改进的方法——希尔排序。

一、插入排序

插入排序的基本思想是把一个个记录按其关键字的大小插入到已经排好次序的记录序列之中，使得插入后的记录序列仍然是有序的。这很象在玩扑克牌时，一边抓牌，一边理牌的过程，抓了一张牌就插到其应有的位置上去。

例如，已知待排序的一组记录关键字的初始序列如下所示：

$$\{49,38,65,97,76,13,27,\overline{49}\}$$

直接插入排序一开始，把第一个记录看成一个有序序列[49]，然后把第 2 个记录按关键字的大小插入到这个有序序列中，成为一个新的有序序列[38,49]。依次类推，一般情况下，第 i 趟直接插入排序的操作为：在含有 i－1 个记录的有序子序列 r[1..i－1]中插入记录 r[i]后，组成一个含有 i 个记录的有序子序列 r[1..i]。在 r[1..i－1]中为 r[i]寻找插入位置的过程可以用顺序查找，从第 i－1 位置开始进行，并且在 r[0]设置监视哨。图 10-1 为直接插入排序算法的框图描述。其 PASCAL 语言描述如下：

```
CONST nmax={最多记录数}；
TYPE node=RECODE
            key:integer;
            data:char;
          END;
   listtype=ARRAY[0..nmax] OF node;
PROCEDURE straisort(VAR r:listtype,n:integer);
  VAR i,j,k:integer;
  BEGIN
    FOR i:=2 TO n DO
```

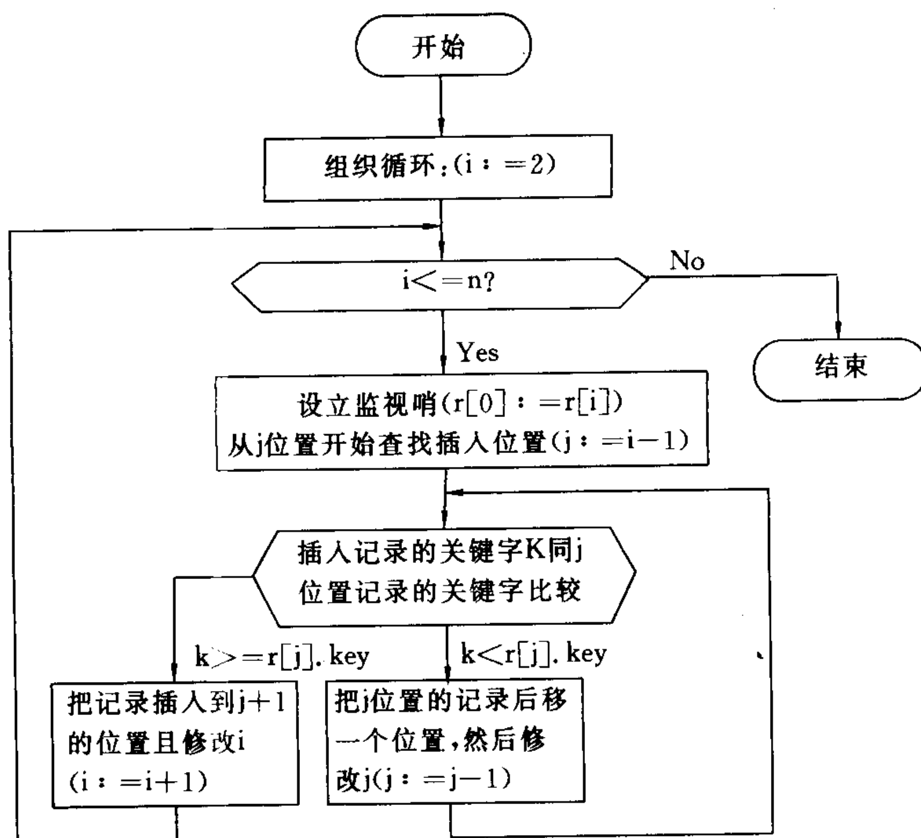


图 10-1　直接插入排序算法框图

```
        BEGIN
          k:=r[i].key; r[0]:=r[i];
          j:=i-1;
          WHILE k<r[j].key DO
            BEGIN
              r[j+1]:=r[j];
              j:=j-1
            END;
          r[j+1]:=r[0]
        END
    END;
```

对上述例子,按照此算法进行直接插入排序的过程如下:

	r[0]	r[1]	r[2]	r[3]	r[4]	r[5]	r[6]	r[7]	r[8]
初始		[49]	38	65	97	76	13	27	$\overline{49}$

i=2	38	[38	49]	65	97	76	13	27	$\overline{49}$
i=3	65	[38	49	65]	97	76	13	27	$\overline{49}$
i=4	97	[38	49	65	97]	76	13	27	$\overline{49}$
i=5	76	[38	49	65	76	97]	13	27	$\overline{49}$
i=6	13	[13	38	49	65	76	97]	27	$\overline{49}$
i=7	27	[13	27	38	49	65	76	97]	$\overline{49}$
i=8	$\overline{49}$	[13	27	38	49	$\overline{49}$	65	76	97]

其中，$\overline{49}$是为了区分前后两个不同的记录，它们的关键字均为 49。

分析上述算法，为了正确地插入第 i 个记录，最多要比较 i 次，最少比较一次，平均比较(i+1)/2 次。如果按平均比较次数计算，则将 n 个记录进行直接插入排序所需的平均比较次数为：

$$\sum_{i=2}^{n}(i+1)/2=(n^2+4n-4)/4\approx n^2/4$$

插入排序中记录的移动次数也是比较多的。为了插入第 i 个记录，需移动记录最多为 i+1 次，最少为 2 次(包括建立监视哨记录)；所需总的平均移动次数近似为 $n^2/4$。

如果将直接插入排序中为记录查找插入位置的方法改为折半查找，则可以减少比较次数，这样的排序方法称为折半插入排序；但不论是直接插入还是折半插入排序，记录的移动次数都是很大的。如果将存储结构由向量改为链表，再进行插入排序，就可以不移动记录，这种方法称为链表插入排序。这几种插入排序方法都是稳定的。

二、希尔排序

希尔排序又称“缩小增量排序”，它属于插入排序的一种，是对直接插入排序方法的改进。

从对直接插入排序的分析得知，其算法的平均时间复杂度为 $O(n^2)$。但是，当待排序记录序列按关键字为“正序”时，其时间复杂度可降低到 O(n)。由此可设想，当待排序记录序列按关键字“基本有序”时，直接插入排序的效率就可大大提高；从另一方面来看，直接插入排序算法简单，并且在 n 值很小时效率也比较高。希尔排序正是从

这两点分析出发对直接插入排序进行改进而得到的一种插入排序方法。其算法思想是：选定一个记录的间隔数 d，把全部记录按此间隔数从第一个记录起进行分组，所有相隔为 d 的记录作为一组，在各组内进行排序；然后减小间隔数，重新分组，再进行组内排序。如此重复，直到间隔数为 1。各组的组内排序可以用直接插入排序，也可以用其它的排序方法。

对间隔的选取，有多种方法。希尔提出的取法是：

$$d_1 = n \text{ DIV } 2 \qquad d_{i+1} = d_i \text{ DIV } 2$$

克努特(Kunth)提出取 $d_{i+1}=d_i/3$。另外，有的人认为 d 取奇数好，有的人则认为 $d_i(i=1、2、\cdots)$ 之间互素好等等；不管如何取，必须不断缩小，最后为 1。下面我们按希尔的方法举例说明：

初始关键字	46	55	20	42	94	17	$\overline{17}$	70
$d_1=4$								
第一趟结果	46	17	$\overline{17}$	42	94	55	20	70
$d_2=2$								
第二趟结果	$\overline{17}$	17	20	42	46	55	94	70
$d_3=1$								
结果	$\overline{17}$	17	20	42	46	55	70	94

从上述排序过程可见，希尔排序的一个特点是子序列的构成不是简单地“逐段分割”，而是将相隔某个“增量 d”的记录组成一个子序列。显然，当 d 较大时，各个子序列的元素较少，使用一些简单的排序方法(如直接插入排序方法)效率也较好；当 d 较小时，由于许多记录已经有序，不需要多少移动，所以就提高了排序的速度，当最后 d 取 1 时，即对整个序列排序。图 10-2 为描述这一算法的框图。

其 PASCAL 语言描述如下：

```
CONST nmax={最多记录数}；
      dn=nmax DIV 2；
PROCEDURE shellsort(VAR r:ARRAY[-dn..nmax] OF node; n:
```

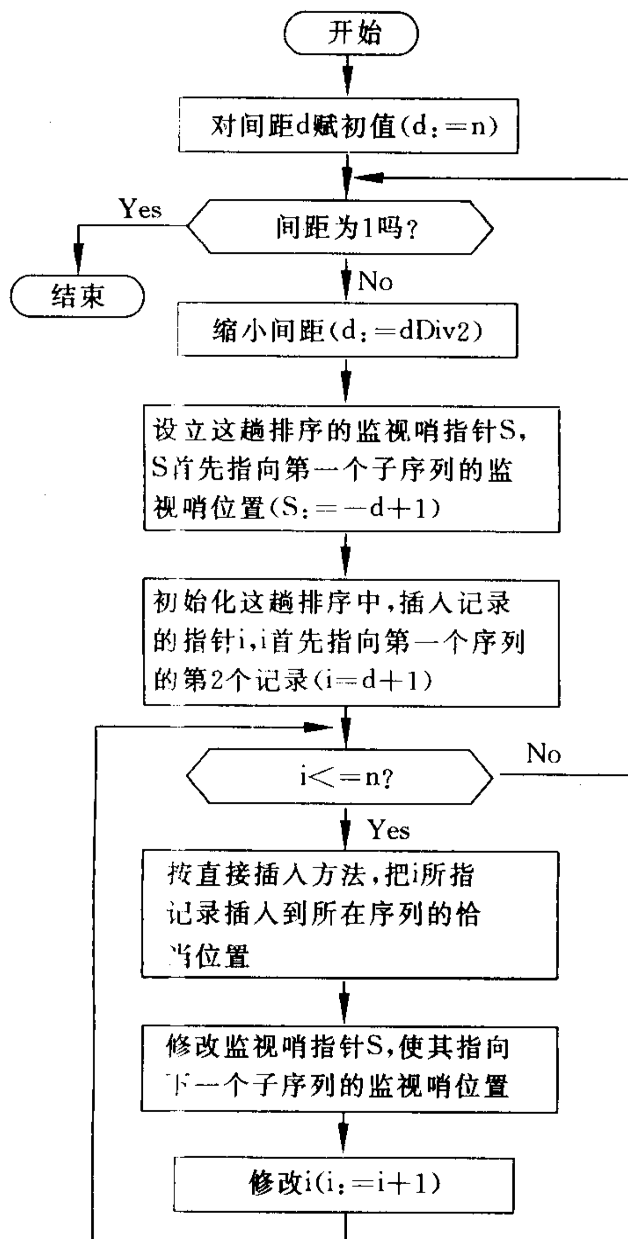


图 10-2　希尔排序算法框图

```
integer);
        VAR d,s,i,j,k:integer;
        BEGIN
          d:=n;
          WHILE d>1 DO
            BEGIN
              d:=d DIV 2
```

```
        s:=-d+1;
        FOR i:=d+1 TO n DO
          BEGIN
            r[s]:=r[i]; j:=i-d; k:=r[i].key;
            WHILE k<r[j].key DO
              BEGIN
                r[j+d]:=r[j]; j:=j-d
              END;
            r[j+d]:=r[s];
            s:=s+1;
            IF s>0 THEN s:=-d+1
          END
      END
    END;
```

希尔排序的速度一般要比直接插入排序快,但具体分析比较复杂,因为它的时间是所取“增量”的函数。希尔排序的平均比较次数和平均移动次数都约为 $n^{1.3}$,有兴趣的读者可以参阅 Kunth 所著的“程序设计技巧”第三卷。希尔排序是不稳定的。

10.2 交换排序

本节中首先介绍基本的交换排序即冒泡排序,然后介绍快速排序。

一、冒泡排序

冒泡排序方法是把记录按纵向排列,然后自下而上地比较相邻记录的关键字 K_j 和 k_{j-1},若 $K_{j-1}>K_j$ 称为逆序,则两者交换位置。再将 K_{j-1} 与 K_{j-2} 进行比较,如有逆序则交换。直至全部关键字均比较一遍。这样一趟加工就使关键字最小的记录上升到第一个位置。然后进行第二趟冒泡排序,对后 n-1 个记录进行同样操作,其结果使关键字次小的记录被安置在第二个位置上。依次类推,设有 n 个记录,

则最多经过 n－1 趟排序后，就使记录按关键字从小到大、自上而下地排好。这个过程就象气泡一个个往上冒一样，故形象地取名为冒泡排序。

例如，有 8 个记录，其冒泡排序过程如下：

初始关键字	第一趟排序后	第二趟排序后	第三趟排序后	第四趟排序后	第五趟排序后	第六趟排序后	第七趟排序后
┌─┐	13	13	13	13	13	13	13
49	┌─┐	27	27	27	27	27	27
38	49	┌─┐	38	38	38	38	38
65	38	49	┌─┐	49	49	49	49
68	65	38	49	┌─┐	$\overline{49}$	$\overline{49}$	$\overline{49}$
76	68	65	$\overline{49}$	$\overline{49}$	┌─┐	65	65
13	76	68	65	65	65	┌─┐	68
27	27	76	68	68	68	68	┌─┐
$\overline{49}$	$\overline{49}$	$\overline{49}$	76	76	76	76	76

在这个例子中，最后五趟排序是多余的，因为，此时记录按关键字已排好序。所以，在算法中应记住每一趟排序时，是否发生过“交换”，若没有发生过交换则说明已排好序，整个排序结束。图 10-3 为冒泡排序算法的框图描述。

这个算法的 PASCAL 语言描述，读者可根据图 10-3 所示框图自已完成。

从冒泡排序过程容易看出，当初始序列为“正序”时，则只需进行

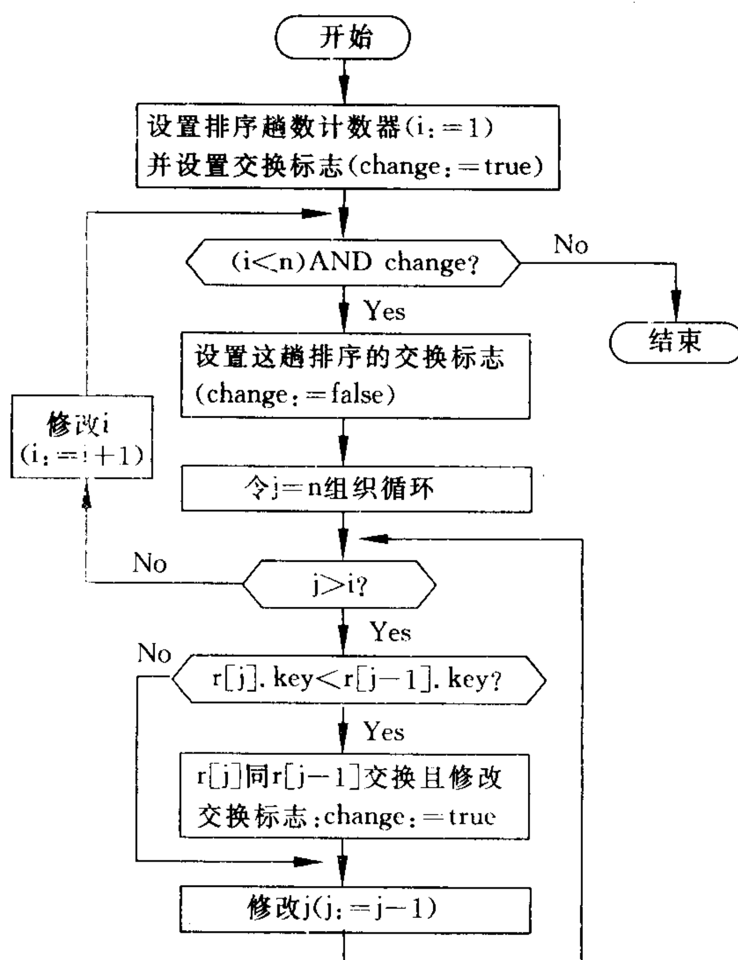


图 10-3　冒泡排序算法框图

一趟排序，在排序过程中只需进行 n－1 次关键字之间的比较，且不需移动记录；反之，若初始序列为“逆序”序列，则需进行 n－1 趟排序，共需进行 n(n－1)/2 次关键字比较以及等数量级的记录交换。因此总的时间复杂度为 $O(n^2)$。冒泡排序是一种稳定的排序方法。

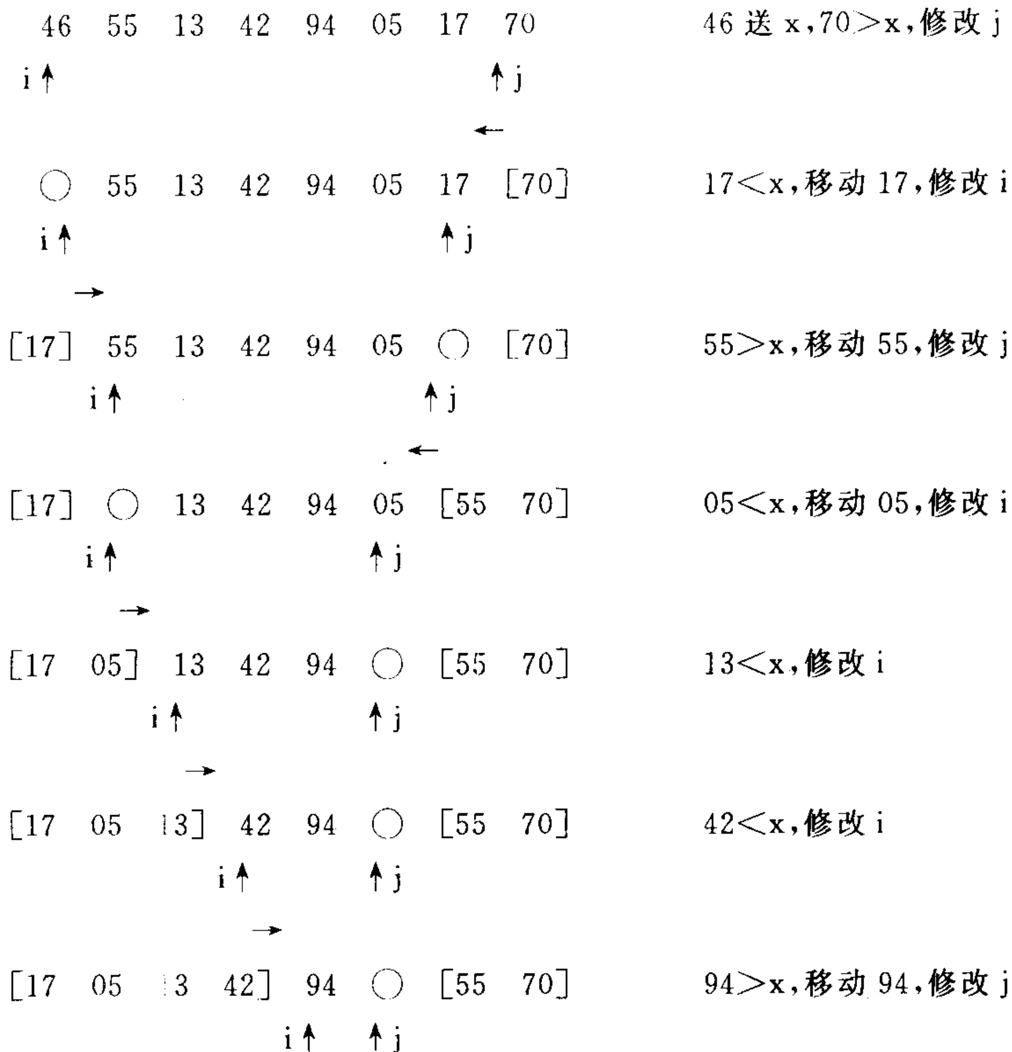
二、快速排序

这种排序方法是由霍尔(Hoare)提出的。这是目前内部排序中速度较快的方法，故称快速排序，其实质是分区交换排序。在这种排序方法中，用一个向量存储 n 个记录，先取向量中第一个记录作为控制记录，设法把该记录放到向量的合适位置上，使得在这个记录右面的所有记录的关键字均大于等于它的关键字，而在它左面的那些记录

的关键字都小于它的关键字，这样就把序列分成了两个子序列，此过程称为一趟快速排序。然后再分别对这两个子序列进行一趟快速排序。依此类推，直至排序完成为止。

那么怎样才能把控制记录放到其合适的位置上呢？假设待排序的序列为{r[s],r[s+1],…,r[t]}，我们可以设立两个位置指针 i 和 j，开始时，令 i=s、j=t。首先取控制记录 x(x:=r[s])，然后：①从 j 所指的位置起向前搜索，找到第一个关键字小于控制记录关键字的记录，将它同控制记录 x 交换位置；②从 i 所指的位置起向后搜索，找到第一个关键字大于控制记录关键字的记录，将它同控制记录 x 交换位置。重复上述两步操作，直到 i=j，则 i 所指示的位置就是控制记录应在的位置。

例如，有一组记录的关键字，其一趟快速排序的过程如下所示：



```
                       ←
[17  05  13  42]  ○  [94  55  70]        i=j,x 送 i 位置
                  i↑ ↑j
[17  05  13  42]  46  [94  55  70]
```

在第一个记录找到合适位置后，就把整个序列分成了左右两部分；再在这两部分上分别重复上述操作。结合上述例子，其分区进行排序的过程如下所示(方括号内为待排序关键字)：

```
[17  05  13  42]  46  [94  55  70]
i↑            ↑j
[13 05]  17  [42]  46  [94  55  70]
i↑   ↑j
 05  13  17  42  46  [94  55  70]
                      i↑       ↑j
 05  13  17  42  46  [70  55]  94
                      i↑   ↑j
 05  13  17  42  46  55  70  94
```

现在我们讨论快速排序算法的具体实现，首先讨论一趟快速排序算法。假设进行一趟快速排序的记录区间的下界和上界分别为 s 和 t，则算法的框图描述如图 10-4 所示。

其 PASCAL 语言描述如下：

```
PROCEDURE qkpass(VAR r:listtype; s,t:integer; VAR i:integer);
VAR j:integer;
    x:node;
BEGIN
  i:=s; j:=t; x:=r[s];
  WHILE (i<j) DO
    BEGIN
      WHILE (i<j) AND (r[j].key>=x.key)
        DO j:=j-1;
      r[i]:=r[j];
      WHILE (i<j) AND (r[i].key<=x.key)
        DO i:=i+1;
```

```
        r[j]:=r[i]
      END;
    r[i]:=x
  END;
```

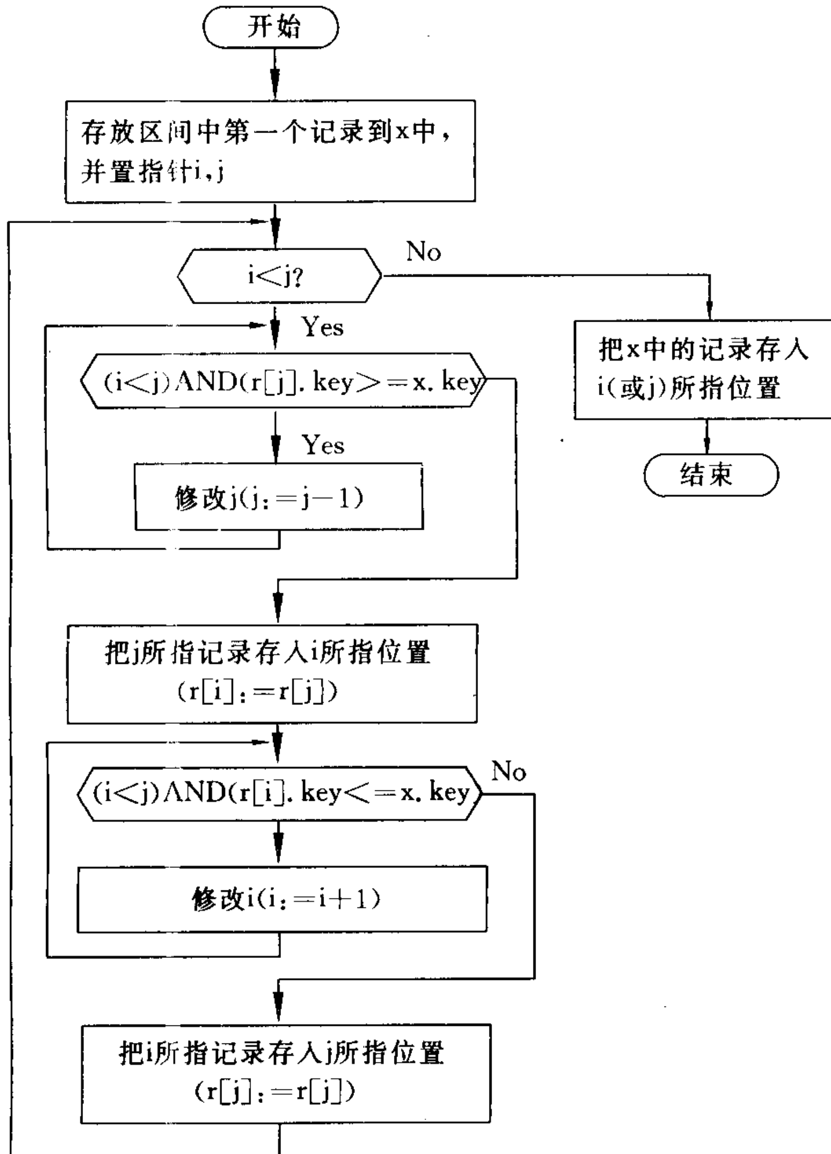


图 10-4 一趟快速排序算法框图

整个快速排序可以写成递归算法，其框图描述如图 10-5 所示，假设待排序的记录区间的下界和上界仍以 s 和 t 表示。

其 PASCAL 语言描述如下：

```
PROCEDURE qksort(VAR r:listtype; s,t:integer);
```

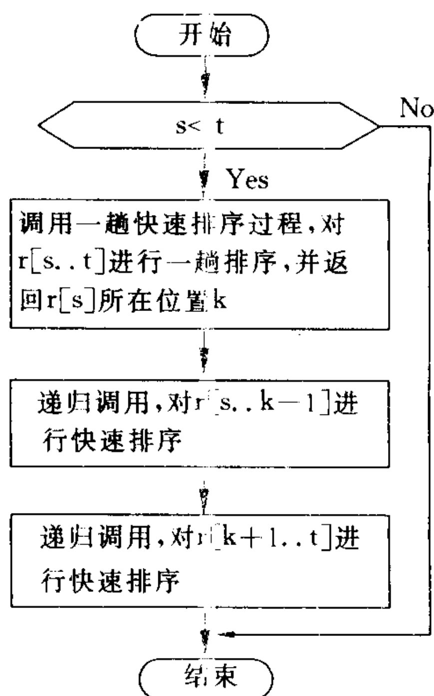


图 10-5　快速排序的递归算法框图

```
VAR k:integer;
BEGIN
  IF s<t THEN
    BEGIN
      qkpass(r,s,t,k);
      qksort(r,s,k-1);
      qksort(r,k+1,t)
    END
END;
```

快速排序的执行时间在待排序的记录按关键字已经有序的情况下为最长。这时，第一趟排序经过 $n-1$ 次比较后，将第一个记录仍定在它原来的位置上，并得到一个有 $n-1$ 个记录的子序列；第二趟排序，经过 $n-2$ 次比较，将第二个记录仍定在它原来的位置上，并得到一个有 $n-2$ 个记录的子序列。依次类推，最后，总的比较次数为 $(n-1)+(n-2)+\cdots+1=n(n-1)/2$，记为 $O(n^2)$。

另一种特殊情况是每趟排序后控制记录的位置正好确定在序列的中央，从而把序列分成大小相等的两个子序列，其总的比较次数为：

$$
\begin{aligned}
T(n) &<= n+2T(n/2) \\
&<= 2n+4T(n/4) \\
&<= 3n+8T(n/8) \\
&\quad \vdots \\
&<= (\log_2 n)n+nT(1)
\end{aligned}
$$

记为 $O(n\log_2 n)$，这是最好情况。可以证明，其平均比较次数也是 $O(n\log_2 n)$。快速排序是一种不稳定的排序方法。

10.3 选 择 排 序

本节将介绍直接选择排序及其改进的方法:堆排序。

一、直接选择排序

直接选择排序的方法是首先在所有的记录中选出关键字最小的记录,将它与第一个记录交换存储位置;然后再在余下的记录中选出关键字次小的记录,将它同第二个记录交换存储位置。依次类推,直至选出所有的记录,使整个序列成为有序序列。下面给出按上述思想进行排序的例子:

初始状态	49	38	65	$\overline{49}$	52	13	27	78
第一趟结果	[13]	38	65	$\overline{49}$	52	49	27	78
第二趟结果	[13	27]	65	$\overline{49}$	52	49	38	78
第三趟结果	[13	27	38]	$\overline{49}$	52	49	65	78
第四趟结果	[13	27	38	$\overline{49}$]	52	49	65	78
第五趟结果	[13	27	38	$\overline{49}$	49]	52	65	78
第六趟结果	[13	27	38	$\overline{49}$	49	52]	65	78
第七趟结果	[13	27	38	$\overline{49}$	49	52	65]	78

直接选择排序算法只要组织一个双循环,描述此算法的框图见图 10-6。

其 PASCAL 语言描述如下。

```
CONST
  max={记录的最多个数};
TYPE
  node=RECORD
         key:integer;
         data:integer
         END;
```

```
sre=ARRAY [1..max] OF node;
PROCEDURE slsort(VAR r:sre; n:
integer);
VAR
  i,j,k:integer;
  x:node;
  BEGIN
    FOR j:=1 TO n-1 DO
    BEGIN
      k:=j;
      FOR i:=j+1 TO n DO
        IF r[k].key>r[i].key
          THEN k:=i;
      IF k<>j THEN
          BEGIN
            x:=r[k];
            r[k]:=r[j];
            r[j]:=x
          END
      END
    END;
```

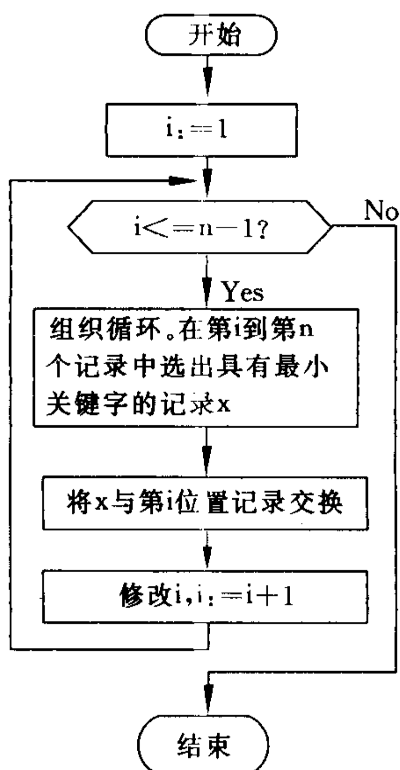


图 10-6 直接选择排序算法的框图

直接选择排序的比较次数与记录的初始排列状态没有关系。第一趟找出最小关键字需 n－1 次比较；第二趟找出次小关键字需 n－2 次比较；依次类推。其总的比较次数为：

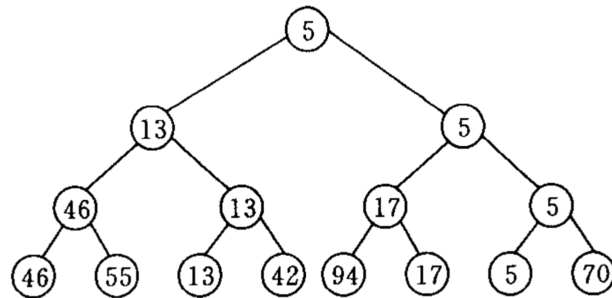
$$\sum_{i=1}^{n-1}(n-i)=n(n-1)/2\approx n^2/2$$

由于每趟选择后，要执行两个记录的位置交换，而这种交换需要用一个暂存空间，所以一次交换实际要进行三次记录的移动操作。上述算法的外循环次数是 n－1 次，所以物理上移动记录的最多次数为 $3(n-1)\approx 3n$。直接选择排序的主要操作是记录间关键字的比较，所以其总的时间复杂度为 $O(n^2)$。直接选择排序是一种不稳定的排序方法。

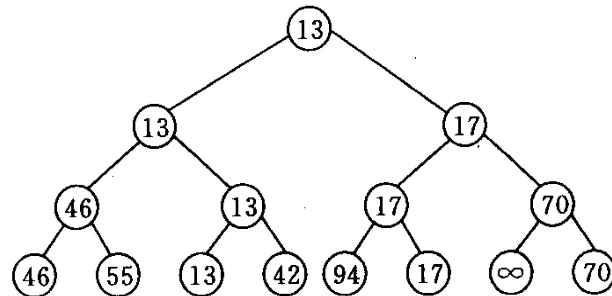
二、堆排序

堆排序(heap sort)是对直接选择排序法的改进。在讨论堆排序之前先介绍一下树形选择排序。

设有一组关键字为{46,55,13,52,94,17,05,70}。通过对关键字的两两分组比较可以形成一棵二叉树,见图 10-7(a)。从图中可以看出,对 n 个记录需进行 n－1 次比较才能选出一个关键字最小的记录,在图 10-7(a)中是经过 7 次比较后才选出 05 的。接着在选关键字次小的记录时就不必再进行 n－2 次比较了。因为根的右子树中的 05 已经选出,它的叶子位置可用∞来代替,再在右子树中进行两次比较选出 17,最后根的左、右孩子比较选出次小关键字为 13 的记录,如图 10-7(b)所示。其间一共只进行了三次比较,即 70 与∞比得 70,70 与 17 比得 17,17 与 13 比得 13。以此方法逐步进行下去,直至全部排序结束。这种方法虽然减少了比较次数,但需要保留许多指针。另外,对 n 个记录需要 2n－1 个存储单元。



(a) 选出最小关键字为05的记录



(b) 选出次小关键字为13的记录

图 10-7　树形选择排序

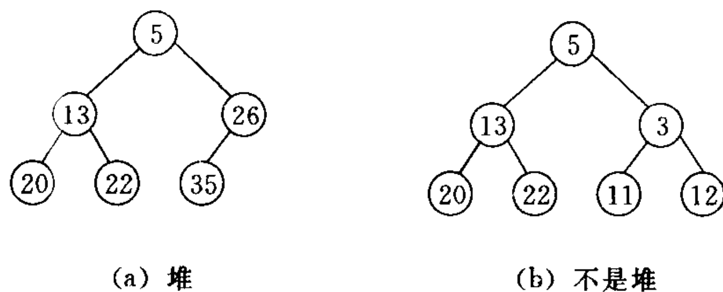
为了克服树形选择排序的缺点，威姆洛斯(J. Willoms)和弗洛伊德(Floyd) 在 1964 年提出了一种改进方法称为堆排序。

堆的定义为：对于一个关键字序列 $K_1, K_2, \cdots, K_n$，当满足如下条件时就称此序列为堆：对一切 i，$1<=i<=\lfloor n/2 \rfloor$，

$$\begin{cases} K_i<=K_{2i} \\ K_i<=K_{2i+1} \end{cases} \quad 或 \quad \begin{cases} K_i>=K_{2i} \\ K_i>=K_{2i+1} \end{cases}$$

我们可以借助完全二叉树来描述堆。如果一棵完全二叉树中，所有结点的值均不大于(或不小于)其左、右孩子的值，则对该二叉树按层次进行遍历时得到的结点序列就是一个堆。在图 10-8 中给出了两棵完全二叉树．其中，图 10-8(a)所示的是一个堆，对此二叉树按层次遍历的结果为：05，13，26，20，22，35；而图 10-8(b)所示就不是一个堆，按层次遍历该二叉树的结果为：05，13，03，20，22，11，12。

从图 10-8(a)可以看出，根结点记录的关键字就是最小关键字。将其输出后，重新建堆则可以得到次小关键字。依次类推，则在堆的定义下，树形选择排序就转化为堆排序了。在示例中虽然按树形来图示，但实际存储时是没有指针的，这些记录被存储在一个向量中。



(a) 堆　　(b) 不是堆

图 10-8　用完全二叉树描述堆

堆是怎样建立起来的呢？可采用所谓的筛选法。

在 n 个记录所对应的完全二叉树中，编号为$\lfloor n/2 \rfloor, \cdots, n$ 的结点都是叶子，已满足堆的定义；所以筛选法可以从编号为 $i=\lfloor 2/n \rfloor$开始；使这棵二叉树中满足堆定义的结点的范围[i..n]不断扩大：i:=i－1；直到 i=1．在每次操作时，可能要对结点的位置进行调整。如果在调整过程中，使下一层已建成堆的子树不再满足堆的定义，则要继续进行调整。这种调整可能会一直延伸到树叶。这种方法好象一层

一层地过筛，每筛一次就将较小关键字往树根方向移动一次。图10-9以完全二叉树形式给出了建堆的示例。其中，初始关键字序列为{46，55，13，42，94，17，05，70}。

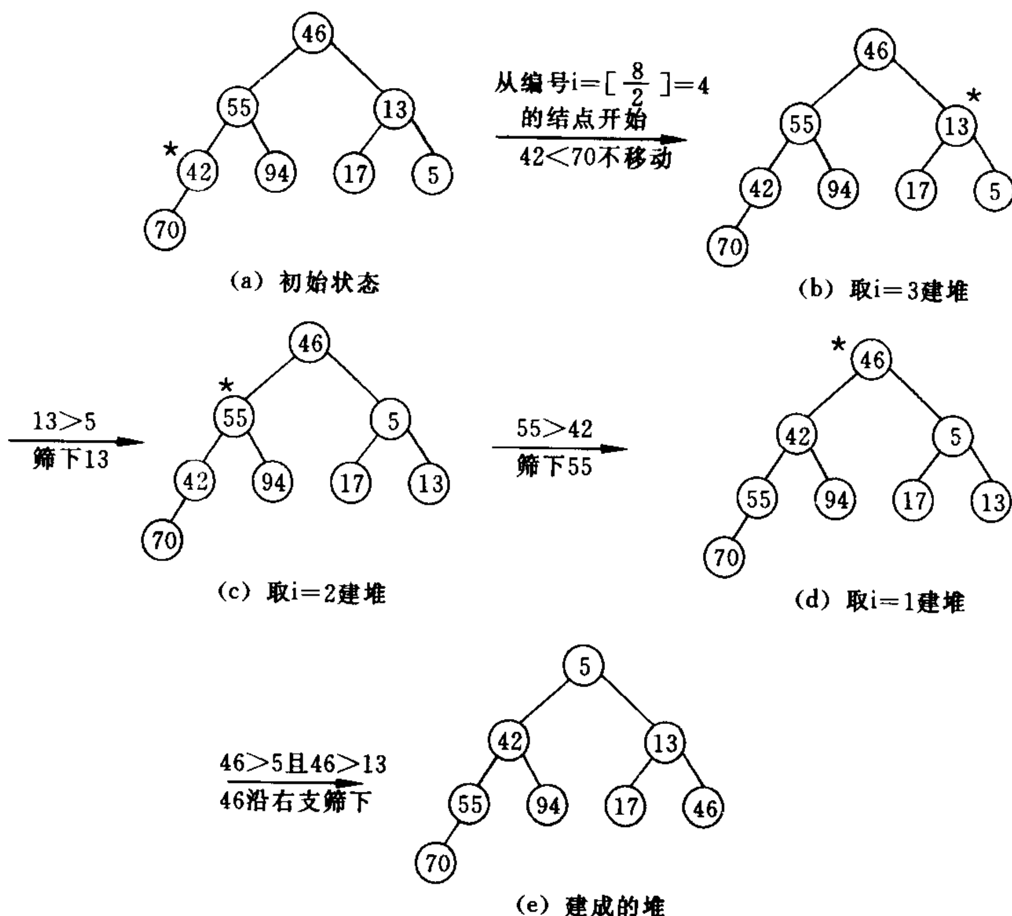


图 10-9 建初始堆过程示例

描述筛选算法的框图如图 10-10 所示。在调用筛选算法时，应给出筛选的范围[s..t]以及存储记录的向量 r。

其 PASCAL 语言描述如下：

```
PROCEDURE heap(VAR r:sre; s,t:integer);
VAR
  i,j:integer;
  x:node;
BEGIN
  i:=s;
  j:=2 * i;
  x:=r[i];
```

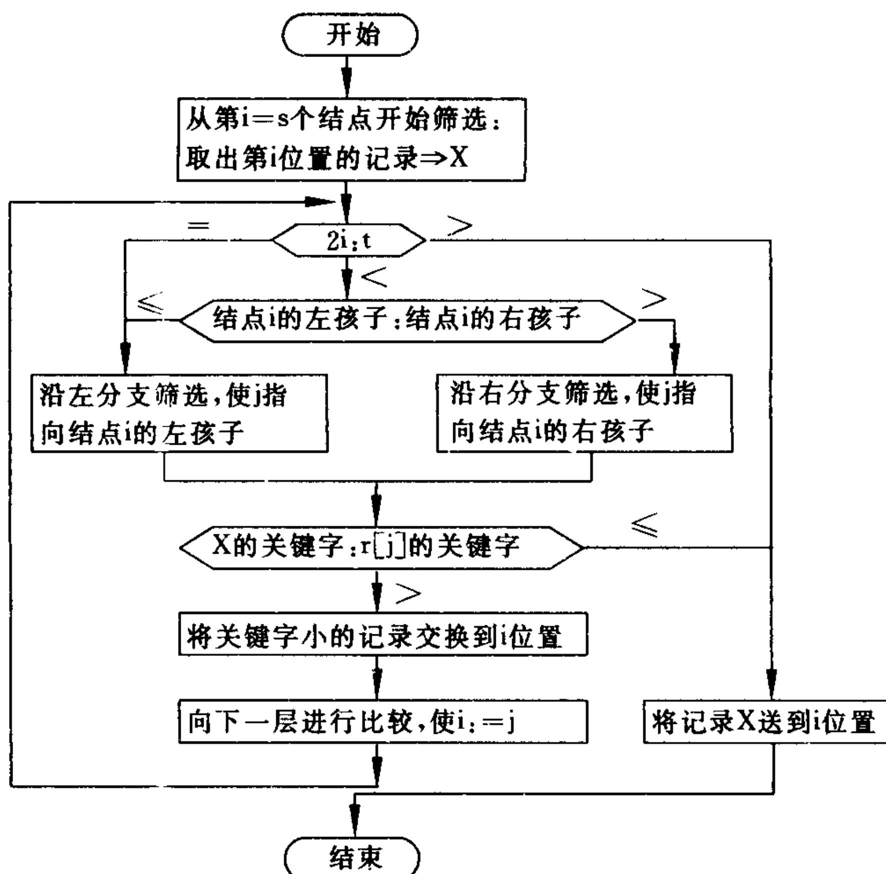


图 10-10　筛选算法框图

```
WHILE j<=t DO
  BEGIN
    IF (j<t) AND (r[j].key>r[j+1].key)
      THEN j:=j+1;
    IF x.key>r[j].key
      THEN
        BEGIN
          r[i]:=r[j];
          i:=j;
          j:=2*i;
        END
      ELSE j:=t+1
  END;
  {WHILE}
r[i]:=x
END;
```

{heap}

初始堆建成后，输出堆顶元素，将最末一个结点移到堆顶位置；然后重新建堆，再输出，再移动。如此反复执行，直到输出全部记录为止。这样就得到了一个记录按关键字的有序序到，从而完成了堆排序。以图 10-9 中的堆为例，在图 10-11 中给出了堆排序的执行过程。

描述堆排序算法的框图如图 10-12 所示。

堆排序算法的 PASCAL 语言描述如下：

```
PROCEDURE heapsort(VAR r:sre; n:integer);
VAR
  temp:node;
  l,k:integer;
BEGIN
  FOR l:=n DIV 2 DOWNTO 1 DO heap(r,l,n)
  FOR k:=n DOWNTO 2 DO
    BEGIN
      write(r[1].data:4);
      temp:=r[1];
      r[1]:=r[k];
      r[k]:=temp;
      heap(r,1,k-1);
    END;
  writeln(r[l].data:4);
END;
{heapsort}
```

堆排序对少量的记录来说，其优越性并不明显，但对大量的记录来说是很有效的。

经分析可以得出堆排序在最坏情况下，其总的计算时间为 $O(n\log_2 n)$。相对于快速排序来说，这是堆排序的最大优点。另外，堆排序仅需一个供交换时暂存记录的辅助空间。它是一种不稳定的排序方法。

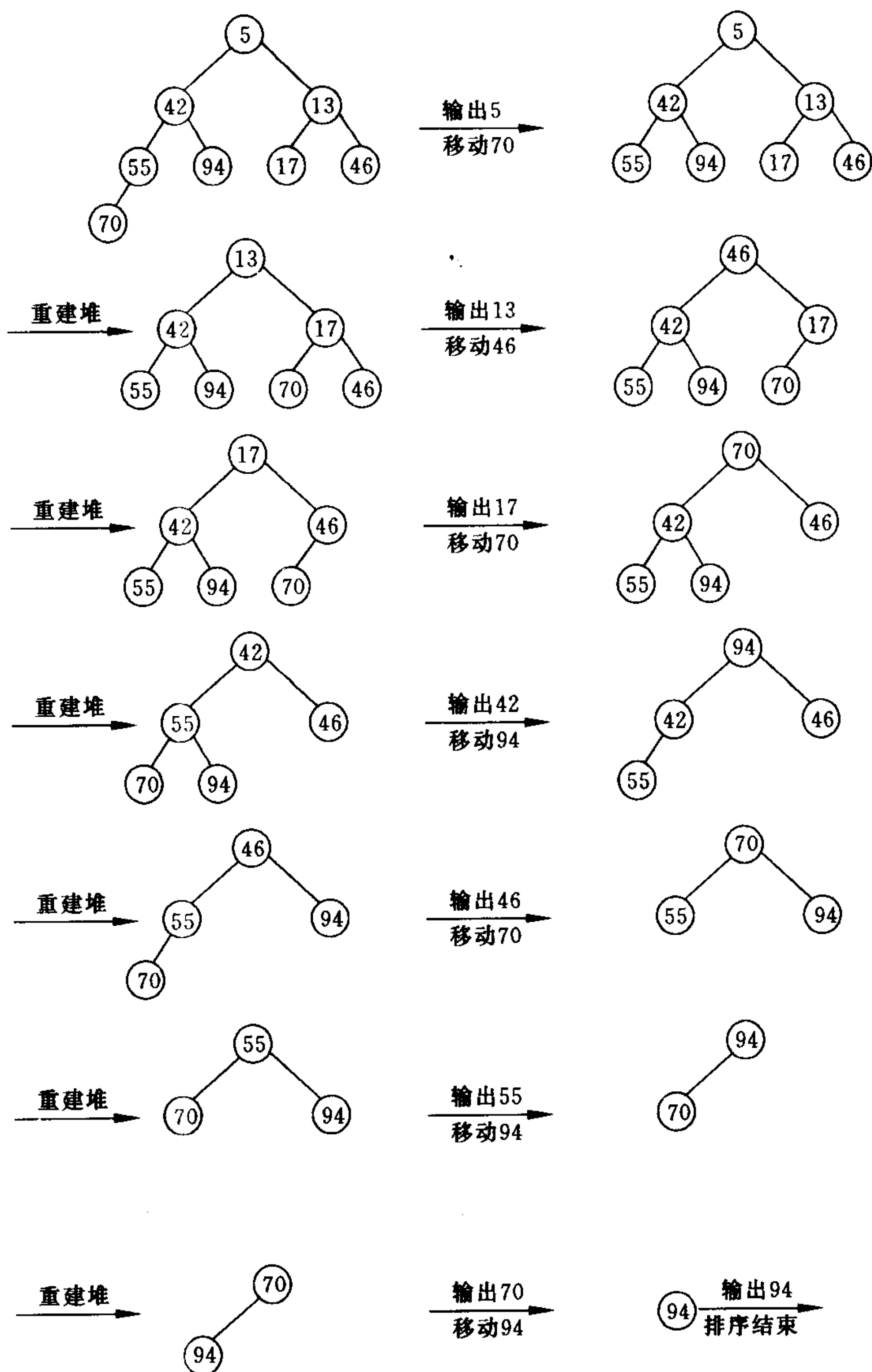

输出5
移动70
重建堆
输出13
移动46
重建堆
输出17
移动70
重建堆
输出42
移动94
重建堆
输出46
移动70
重建堆
输出55
移动94
重建堆
输出70
移动94
输出94
排序结束
输出序列:5,13,17,42,46,55,70,94

图 10-11　堆排序示例

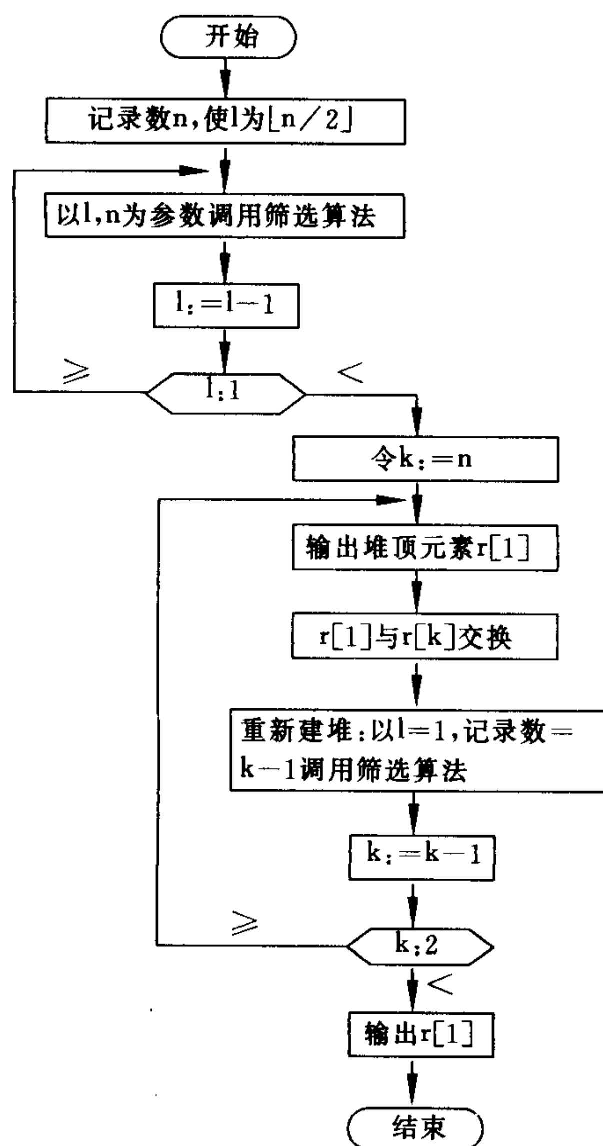


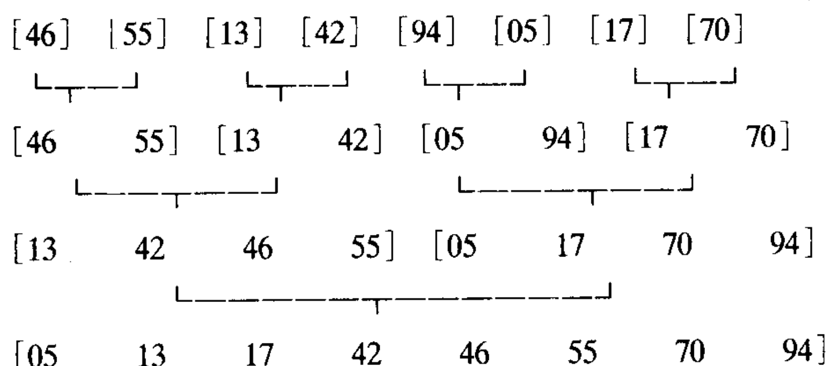
图 10-12　堆排序算法框图

10.4　归 并 排 序

归并排序是另一种类型的排序方法。归并(merging)的意思是把两个或两个以上的有序序列合并起来,形成一个新的有序序列。开始时,可把一个有 n 个记录的无序序列看成为 n 个只一个记录的有序子序列,然后进行两两归并,最后形成一个有 n 个记录的有序序列。

例如,给出有八个记录的某序列,其关键字分别为:46,55,13,

42,94,05,17,70。归并时，先把这个序列看成八个长度为 1 的有序子序列，然后逐步进行归并，排序过程如下所示：



上述排序方法中总是反复将两个有序序列归并成一个有序序列，所以称为两路归并排序。

上述例子中，经过三趟归并完成了排序。第一趟归并时，取归并长度 l=1(即子序列中的记录数为 1)。在一趟归并中要进行多次的两两归并，分别使长度为 l 的首尾相连的二个子序列归并为一个有序序列。然后，长度增加一倍再进行第二趟归并。依次类推，直至排序结束。

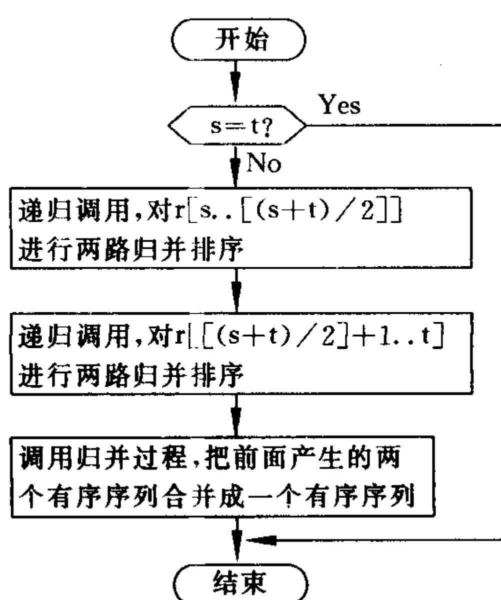


图 10-13　两路归并排序的递归算法框图

递归形式的两路归并排序算法的框图描述如图 10-13 所示。此算法实现对 r[s..t]中的记录进行两路归并排序，结果使 r1[s..t]中记录按关键字有序，其中用到了一个辅助向量 r2[s..t]用于暂存归并过程中的记录。

这个算法的 PASCAL 语言描述如下：

```
PROCEDURE mergesort(VAR r,r1:listtype;s,t:integer);
VAR m:integer;
```

```
    r2:listtype;
BEGIN
  IF s=t THEN r1[s]=r[s]
  ELSE BEGIN
         m:=(s+t) DIV 2;
         mergesort(r,r2,s,m);
         mergesort(r,r2,m+1,t);
         merge(r2,s,m,t,r1);
       END
END;
```

在两路归并排序算法中，我们调用了一个两两归并的算法 merge(r1,l,m,n,r2)，它的功能就是将在 r1 中的两个有序序列 r1[l..m]和 r1[m+1..n]合并成一个有序序列，并存放在 r2[l..n]中。这个算法的框图描述如图 10-14 所示。

这个算法的 PASCAL 语言描述如下：

```
PROCEDURE merge(r1:listtype;l,m,n:integer;VAR r2:listtype);
VAR i,j,k,p:integer;
BEGIN
  i:=l;j:=m+1; k:=l;
  WHILE (i<=m) AND (j<=n) DO
    IF r1[i].key<=r1[j].key
      THEN BEGIN
             r2[k]:=r1[i];
             i:=i+1;
             k:=k+1
           END
      ELSE BEGIN
             r2[k]:=r1[j];
             j:=j+1;
             k:=k+1
           END;
  FOR p:=i TO m DO
```

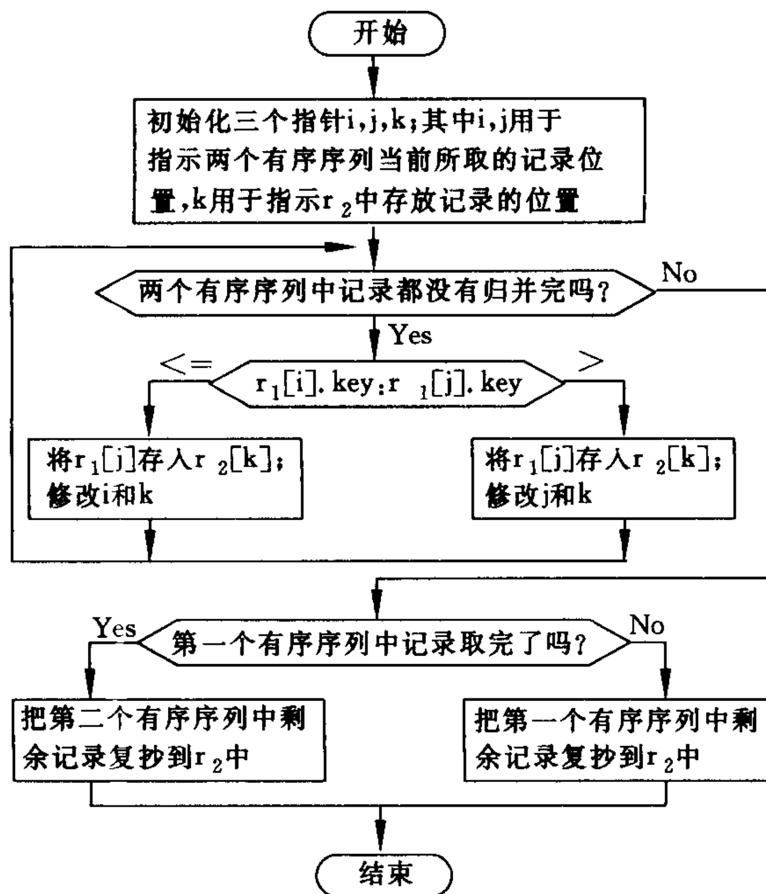


图 10-14　两两归并算法框图

```
    BEGIN
      r2[k]:=r1[p];
      k:=k+1
    END;
  FOR p:=j TO n DO
    BEGIN
      r2[k]:=r1[p];
      k:=k+1
    END
END;
```

两路归并排序总的时间复杂度为 $O(n\log_2 n)$，并且需要同原存储空间大小相等的辅助存储空间，它是一种稳定的排序方法，这个排序方法也可用于外部排序。

10.5 基 数 排 序

基数排序(radix sort)的设计思想与以前的各种排序方法均不相同,它是按组成关键字各位的值进行排序的。

基数排序把一个关键字 K 看成一个 d 元组,即:$K_1,K_2,\cdots,K_d$。其中,$0<=K_j<r$,这里的 r 称为基数。若关键字是十进制的整数,则 r=10;若关键字是八进制数,则 r=8。d 表示关键字的位数,此位数按所有待排序的关键字中最长的一个进行计量,其他不足 d 位的关键字则在前面补零。

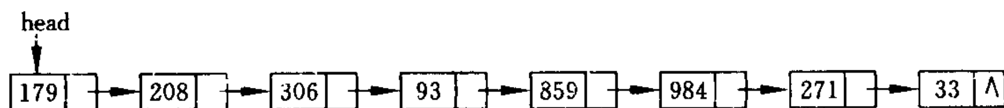
在 $K_1,K_2,\cdots,k_d$ 中,K_1 称为最高有效位,K_2 称为次高有效位,K_d 称为最低有效位。基数排序入可以从最高有效位开始,出可以从最低有效位开始。现在仅讨论从最低有效位开始的方法。

基数排序的基本思想是设立 r 个队列,首先按关键字最低有效位的值,把 n 个记录分配到这 r 个队列中;然后将各队列中的记录依次收集起来。这时,n 个记录已经按关键字最低有效位的值从小到大排好了次序。接着,再按关键字次低有效位的值把刚收集起来的记录依次再分配和收集,直至关键字的最高有效位。这样就得到了一个按关键字从小到大有序的记录序列。为了减少记录的移动次数,队列采用链式存储结构,每个链表(队列)i 设立两个指针,一个指向队头,即指向第一个进入该队列的记录,记为 F[i];另一个指针指向队尾,即指向队列中刚进入的记录,该指针记作 E[i]。现举例说明如下:设有 8 个记录,其关键字分别为 179,208,306,093,859,984,271,033。它们是十进制数,所以基数 r=10,关键字的位数 d=3。这一组记录的基数排序过程如图 10-15 所示。

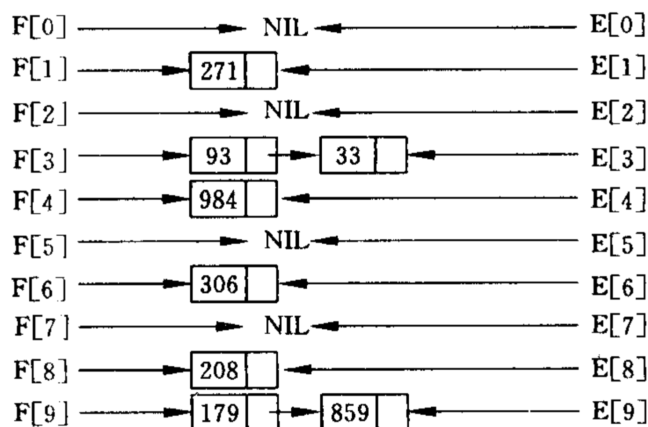
图 10-16 为基数排序算法的框图描述。在此,假设存储记录的链表表头为 head 以及基数为 r,组成关键字的位数为 d。

这个算法的 PASCAL 语言描述如下:

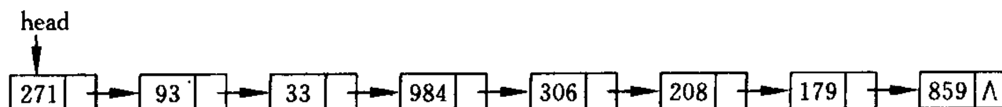
```
CONST
  maxd={关键字的位数最大值};
```



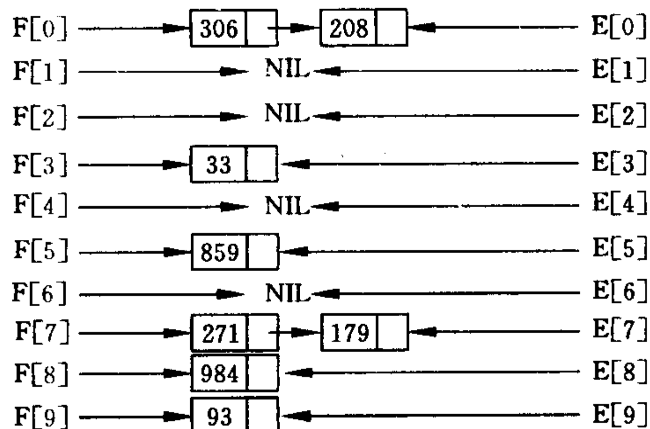
(a) 初始状态



(b) 第一次按最低有效位分配后各队列的情况

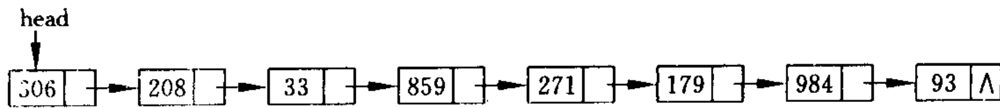


(c) 第一次按最低有效位收集后的情况

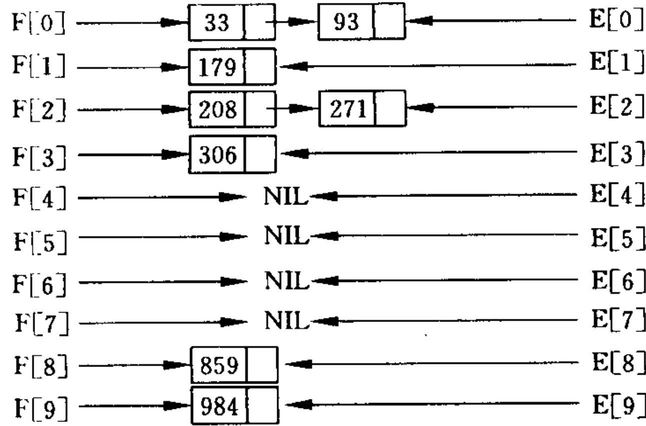


(d) 第二次按次低有效位分配后各队列的情况

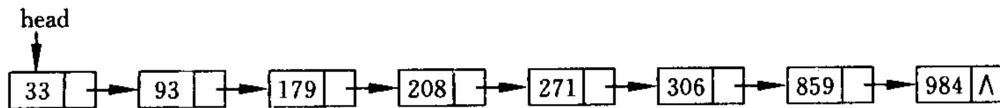
```
  radix={基数};
TYPE
  redlink=↑rednode;
  rednode=RECORD
            key:ARRAY [1..maxd] OF 0..radix-1;
            data:char;
            next:redlink;
```



(e) 第二次按次低有效位收集后的情况



(f) 第三次按最高有效位分配后各队列的情况



(g) 第三次按最高有效位收集后的情况

图 10-15 基数排序示例

```
            END;
    arrtp=ARRAY [0..radix-1] OF redlink;
  PROCEDURE vadixsort(VAR head:redlink; r,d:integer);
  VAR k,i,j:integer;
        p,q:redlink;
        f,e:arrtp;
  BEGIN
    FOR i:=d DOWNTO 1 DO
      BEGIN
        p:=head;
        head:=NIL;
        j:=0;
        REPEAT
          f[j]:=nil;
          e[j]:=nil;
```

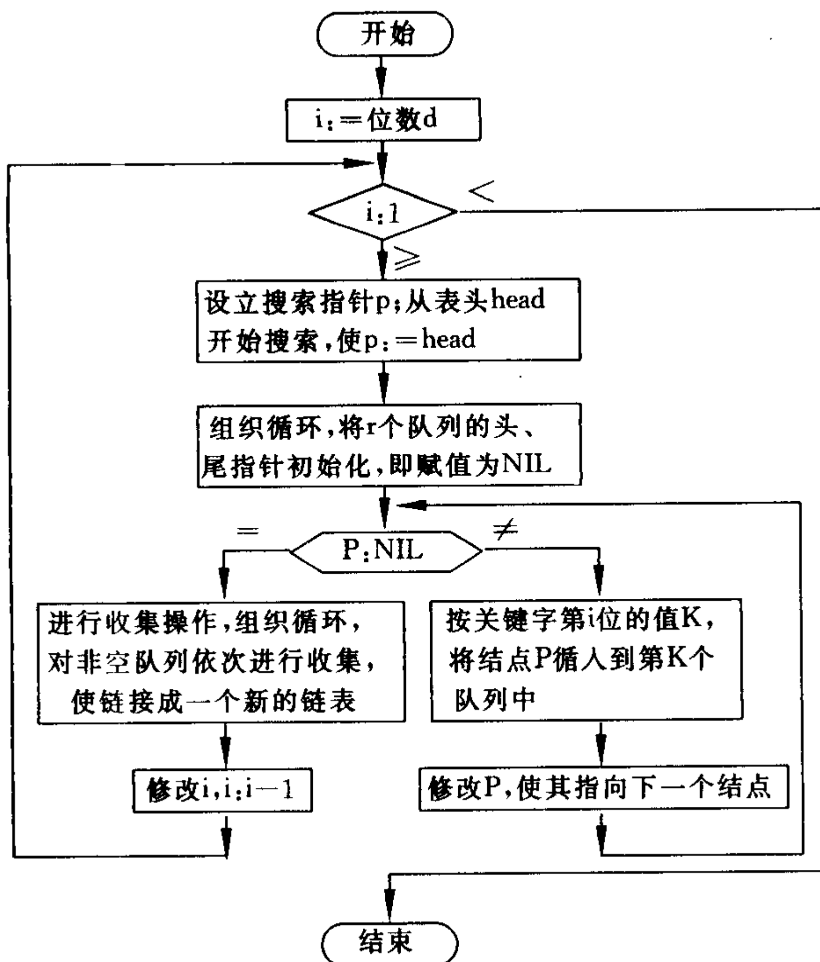


图 10-16 基数排序算法框图

```
    j:=j+1;
  UNTIL j=r;
  WHILE p<>nil DO
    BEGIN
      k:=p↑.key[i];
      IF f[k]=nil
        THEN f[k]:=p
        ELSE e[k]↑.next:=p;
        e[k]:=p;
      p:=p↑.next;
    END;
  j:=0;
```

```
              REPEAT
                IF f[j]<>nil
                  THEN
                    BEGIN
                      IF head<>nil
                        THEN q↑.next:=f[j]
                        ELSE head:=f[j];
                      q:=e[j]
                    END;
                j:=j+1
              UNTIL j=r;
            q↑.next:=nil
            END
        END;
```

上述程序中,f 和 e 分别是队列的首、尾指针,r 为基数,d 为关键字的位数。在收集操作中,设立了一个指针 q,它总是指向当前新形成的链表的最后一个结点。设立 q 指针的目的是为了能够快速、方便地将每一个非空队列链接到当前链表的后面。值得一提的是在分配、收集操作中,实际上并没有进行记录的移动,而仅仅是修改有关的指针,从而形成按关键字最低几位的数值大小排序的链表。

分析上述算法,对于 n 个记录,执行一次分配和收集的时间为 $O(n+r)$。如果关键字有 d 位,则要执行 d 遍。所以总的计算时间为 $O(d(n+r))$。可见对于不同的基数 r 所用的时间是不同的。当 r 或 d 较小时,这种方法较为节省时间。基数排序所要求的附加存储量是 r 个队列的头、尾指针,即为 2r 个存储单元;由于待排序记录是以链表形式存储的,故相对于顺序存储结构而言,还增加了 n 个指针域的空间。

以上我们介绍了一些常用的内排序方法,现将它们的优缺点作一简单的比较。

通常从以下两个方面来衡量排序方法的优劣:

1. 排序期间的计算工作量,一般考虑关键字的平均比较次数和

记录的平均移动次数。

2. 排序期间所需的附加存储容量。

本日中介绍的排序方法，基本上可以分为两大部分。一部分是简单的直接算法，即直接插入排序，直接选择，冒泡排序等；另一部分是上述方法的改进，如希尔排序，快速排序和堆排序等。直接算法的计算量一般为 $O(n^2)$，而它们的改进算法则为 $O(n\log_2 n)$。但这些估算都是很粗略的，而且没有考虑其他的诸如循环控制等计算量。所以，一些研究者通过实验来获得较为准确的计算时间。如 N. Wirth 的“算法＋数据结构＝程序”一书以及 D. E. Knuth 的“计算机程序设计技巧”(第三卷)中都给出了实验数据。目前，较为一致的结论是：快速排序方法最快，其次是归并排序，堆排序和希尔排序。当待排序序列的记录数较少(如记录数＜＝25)时，直接插入排序是较为有效和实用的。

从存储容量来看，归并排序需要的附加存储量最大，对 n 个记录需要附加一倍的存储量。其次是基数排序，它需要附加较多的存储单元用于存储队列指针。至于快速排序，附加的存储单元是用于调用时存放指针的栈，栈的大小取决于递归的嵌套深度。

总的来说，每种排序方法都有它本身特殊的优点，所以有实际应用中，由于应用场合所给数据结构的不同，很难确定哪一种方法是最好的。因此，对排序方法的选择要因地制宜。

习　　题

1. 试编写在链表存储结构上实现插入排序和选择排序的算法。
2. 给出一组待排序的记录，其关键字为：12，2，16，30，8，28，4，10，20，6，18。写出用下列方法进行排序时，每一趟排序时的状态：

   (1) 快速排序 (2) 希尔排序 (3) 堆排序 (4) 归并排序 (5) 基数排序
3. 判别以下序列是否为堆。如果不是，则把它调整为堆：

(1) (186, 86, 48, 73, 35, 39, 42, 57, 66, 21);

(2) (18, 70, 33, 65, 24, 56, 48, 92, 86, 33);

(3) (208, 167, 56, 38, 66, 23, 42, 12, 30, 52, 8, 20);

(4) (6, 56, 20, 23, 40, 38, 29, 61, 35, 76, 28, 168)。

4. 已知(k_1, k_2,…, k_n)是堆，试编写一个算法将(k_1, k_2,…, k_n, k_{n+1})调整为堆。

5. 试编写折半插入排序算法。

参 考 文 献

[1] 严蔚敏、吴伟民,《数据结构》,清华大学出版社,1992 年。

[2] 潘道才,《数据结构》,成都电讯工程学院出版社,1988 年。

[3] 梁晋清、施振夏、陆菊康,《程序设计 PASCAL 》,上海交通大学出版社,1992 年。

[4] 沈石山、俞鑫泰,《集合论初步》,上海教育出版社,1980 年。